Fall 12-22-2015

Formal Modeling and Verification of Delay-Insensitive Circuits

Hoon Park
Portland State University

Formal Modeling and Verification of Delay-Insensitive Circuits

by

Hoon Park

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Dissertation Committee:
Xiaoyu Song, Chair
Douglas V. Hall
Fu Li
Christof Teuscher
Jingke Li
Marly Roncken (non-voting member)

Portland State University
2015

Abstract

Einstein's relativity theory tells us that the notion of simultaneity can only be approximated for events distributed over space. As a result, the use of asynchronous techniques is unavoidable in systems larger than a certain physical size. Traditional design techniques that use global clocks face this barrier of scale already within the space of a modern microprocessor chip. The most common response by the chip industry for overcoming this barrier is to use Globally Asynchronous Locally Synchronous (GALS) design techniques. The circuits investigated in this thesis can be viewed as examples of GALS design. To make such designs trustworthy it is necessary to model formally the relative signal delays and timing requirements that make these designs work correctly. With trustworthy asynchrony one can build reliable, large, and scalable systems, and exploit the lower power and higher speed features of asynchrony.

This research presents ARCtimer, a framework for modeling, generating, verifying, and enforcing timing constraints for individual self-timed handshake components that use bounded-bundled-data handshake protocols. The constraints guarantee that the component's gate-level circuit implementation obeys the component's handshake protocol specification. Because the handshake protocols are delay insensitive, self-timed systems built using ARCtimer-verified components can be made delay insensitive. Any delay sensitivity inside a component is detected and repaired by ARCtimer. In short: by carefully considering time locally, we can ignore time globally.

ARCtimer applies early in the design process as part of building a library of verified components for later system use. The library also stores static timing analysis (STA) code to validate and enforce the component's constraints in any

self-timed system built using the library. The library descriptions of a handshake component's circuit, protocol, timing constraints, and STA code are robust to circuit modifications applied later in the design process by technology mapping or layout tools.

New contributions of ARCtimer include:

1. Upfront modeling on a component by component basis to reduce the validation effort required to

   (a) reimplement components in different technologies,

   (b) assemble components into systems, and

   (c) guarantee system-level timing closure.

2. Modeling of bounded-bundled-data timing constraints that permit the control signals to lead or lag behind data signals to optimize system timing.

Dedication

This dissertation is dedicated to my wife Haera, for all her love and support.

Acknowledgments

First of all I thank my mentor, Marly Roncken. She is enthusiastic, full of ideas, and thorough with all the details. I appreciate all her contributions in time, ideas, and funding to make my Ph.D. experience productive and stimulating. The joy and enthusiasm she has for research was contagious and motivational for me, even during tough times in the Ph.D. pursuit.

I thank my advisor, Prof. Xiaoyu Song, for his guidance, encouragement, and excellent scientific advice throughout the course of this research.

Special thanks go to Willem Mallon who built the ARCwelder compiler during the two years he joined the ARC at Portland State University. He introduced me to the theory of Delay-Insensitive algebra and developed the notion of Bounded Bundled Data (BBD), which I have formalized in this thesis.

I thank Anping He, our ARC collaborator in China, with whom we have weekly meetings. He was very instrumental in building a foundation for automating ARCtimer.

I would also like to thank my fellow ARCwelders: Ivan Sutherland for stimulating ideas and discussions, and Swetha Mettala Gilla, Chris Cowan, and Navaneeth Jamadagni, for daily companionship and laughter.

Most of all, I thank my beloved wife, Haera Chung, for her love, constant support, her prayers, and giving birth to our son, Daniel. Daniel is now five months old at 18 pounds – he is a bundle of joy.

Last but not least, I would like to thank my parents, Woe-Chul Park and Kum-ju Kim for their unending love and support. It's been a long journey.

Table of Contents

List of Figures

1

Introduction

1.1 Motivation

Modern computer systems are distributed over space. For example, there is the internet of things – a network of physical objects embedded with electronics, software, sensors, and network connectivity, which enables these objects to collect and exchange data. Another example is IBM's TrueNorth [28]. TrueNorth is a system composed of modular chips that act like neurons and form artificial neural networks to run "deep learning algorithms" like Skype's chat translator or Facebook's facial recognition.

Global state is a useful model for traditional clocked hardware, where: state may change only when the clock ticks, where all tasks must fit into the clock period, and where the global state is stable between ticks.

Below folow three examples where events are not simultaneous over space.

The first example relates to the latest planetary mission to Pluto. New Horizons took a picture of Pluto and its moon, Charion. Pluto is very far away – more than 30 times Earth's distance from the Sun. It took about 4 hours to tweak the position of New Horizons to take such a picture. There's a substantial communication delay between earth and Pluto. Clearly, the commands and the photo shot are not simultaneous.

The second example is also from outer space. In August 2012, NASA landed Curiosity on Mars. At the time, Mars was a communication distance of 13 minutes away. During the last 13 minutes of landing, the landing system worked on

autopilot – self-directed, self-controlled, autonomous.

As a third example, let's take a look at something closer and more down to earth. What about chips? What about communication delays within a cubic inch? Even here, down to earth, within a cubic inch of silicon, the communication delay in a network chip with a size of less than a cubic inch are longer than the delays that we use to control the chip. Moreover, the complexity of pretending that events in the chip are simultaneous is huge: hundreds of clock domains, and ten thousands of clock synchronizers [58].

Clearly, the notions of global state and global control fail to scale over space, even within the cubic inch space of a single chip.

With the advance of IC manufacturing technologies, high speed digital systems have grown in complexity. In the past, all digital systems required data to pass sequentially through the system. The standard approach to this was to synchronize the entire system to a common global discrete period clock. This results in synchronous systems that are orchestrated by a centralized clock that operates on a fixed rate. Control and data signals are stored and passed in lockstep on fixed intervals as determined by the clock and its phases. All functions between the storage elements are evaluated during the clock period. This is a valid model as long as the clock period is longer than the time it takes for the clock signal to travel from one end to another end of the circuit, giving a well-defined semantics to the term 'lockstep.'

Thanks to continuous scaling of VLSI to meet the ever-increasing demands for more speed and less power, the clock period has shrunk to below the chip's end-to-end clock traversal time. In addition, in many high-performance designs, clock power has been reported to exceed 30% of the total power consumption [10].

Power is now an additional limiting factor in raising the clock frequency [57]. As a result, the conceptual framework of synchronous design faces new challenges beyond pure functionality and raw throughput, concerning timing closure and power dissipation, process variation and interfacing.

Due to these drawbacks, asynchronous circuits are gaining interest. Asynchronous circuits, also known as self-timed circuits, do not have a global clock. The asynchronous circuits or self-timed circuits communicate with their neighbors through handshake protocols.

Potential advantages of self-timed circuits compared to synchronous circuits are as follows:

- High speed

  Since there is no central clock, there are no clock skew problems. Self-timed circuits can be designed for average performance while synchronous circuits are typically designed for worst case performance.

- Low power

  Self-timed circuits consume power when and where needed. In contrast, synchronous circuits usually have a ticking clock that consumes power even if there is no work to do.

- Robustness

  Self-timed circuits self-adapt to temperature and voltage variation. In contrast, synchronous circuits require clock frequency regulations to track voltage variations.

- Low electromagnetic interference

  The clock pulses on synchronous circuits operating on the exact same frequency generate resonance and electro-magnetic interference (EMI). The irregular behavior in self timed circuits provides very low EMI.

- Modularity

  The asynchronous or self-timed components communicate with each other through handshake protocols. The protocols are implemented locally, with local timing constraints. The locality of time make it possible to build large systems of any scale.

However, there are also drawbacks such as:

- Different notion of time

  The global clock is replaced by handshake protocols.

- Testability and Debug

  Synchronous circuits can simply freeze the clock for better visibility of circuit actions. We recently provided a generic solution to freeze local self-timed actions for test and debug with *Mr.GO* [39].

- Lack of EDA tool support

  Lack of CAD tool support from EDA industries makes wide adoption of self-timed design hard. The ARCtimer presented in this thesis shows how to provide EDA tool support for timing closure of self-timed systems.

The biggest problems are the lack of a standard work flow, and a non-standard way of designing circuits. A difference in the timing regime requires a different

way of thinking of circuit design, which isn't being offered as education in many places. Portland State University is one of the research institutes world wide where students can study self-timed design. Some companies that have adopted self-timed circuit designs have their own design flow and tools which are not publicly available.

This thesis tackles the problem of lacking a standard workflow by providing a general methodology and work flow outline for solving the local timing constraints that support the handshake protocols between self-timed components. This workflow is called ARCtimer.

The near-term goal of ARCtimer is to generate and repair local timing constraints necessary for correct communication through handshaking. Once the handshake protocols can be assumed correct, the remaining and recurring task of building systems out of handshaking circuit components becomes a delay-insensitive design task. Thus, the longer-term goal of ARCtimer is to support the large-scale integration of delay-insensitive circuits. Hence, the title of this thesis.

1.2 Problem Definition

In digital circuits, a logic hazard is an unexpected output that lasts temporary after an input has changed. Because hazards are temporary, synchronous circuits can usually clear the hazard by slowing down the clock cycle used for synchronizing such that the correct output is in the right place by the time the active clock edge arrives. Self-timed circuits on the other hand does not have a global clock to delay the sampling of the output. Each component operate on their own pace solely relying on handshake signals to communicate with neighboring components.

Because handshake operations are driven by events, glitches must be avoided.

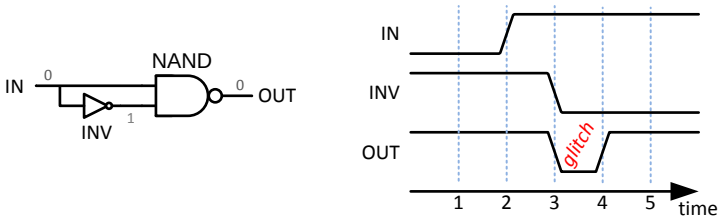


Figure 1.1: An example of combinational logic that could cause a glitch. From the initial state as marked in the Figure, once the IN makes a rising transition ($time = 2$), INV makes a falling transition ($time = 3$). The NAND gate seeing both inputs high (between $time = 2, 3$) produces a momentarily glitch ($time = 3$), but then recovers (after $time = 4$).

To make a circuit free of hazards, we look at *delay-insensitive* (DI) circuits which are the most robust of the asynchronous circuit classes. This circuit class operates correctly with unknown delays in wires and gates. But due to heavy restrictions, only a few circuits are truly insensitive to delays of gates and wires. Timing constraints can be applied to make the circuit operate in a DI fashion, but how can one be sure that the timing constraint set is complete? To verify these timing constraints, what kind of tools should one use, and how would one model the system? What kind of timing constraints should one add? Are the constraints placed at optimal locations?

The problem with lack of support from the industry and standardized tools makes it hard to widely adopt self-timed designs. The solutions are not interchangeable and usually don't carry over with technology advancements. *Static timing analysis* (STA) loop cutting doesn't work with conventional STA tools because they are built for synchronous circuits.

1.3 Proposed Approach

The circuit can behave like a delay-insensitive (DI) circuit if each handshake component faithfully follows their specified DI protocols. ARCtimer uncovers the delay sensitivities and defines a delay repair procedure in advance, so that when the component is used the designer and the STA tool know what to analyze and how to repair it, if it fails DI.

We design our circuits using the theory of Logical Effort [56], where we assume gates and wires are well-designed and there are no adverse analog effects. For the handshake component's protocol, we use the formalism of Delay-Insensitive Algebra developed by [14, 20, 59], which has a compact and complete specification. Delay-Insensitive Algebra specifies both safety property and liveness property, which are important for choices of action.

To verify our design, we use a general purpose model checker to perform an exhaustive verification. Each of the component and protocol specification are modeled for the model checker. The properties such as safety and liveness properties come from the protocol specification, and with the help of a model checker, we find a complete set of timing constraints that are required to make the circuit behave like a delay-insensitive circuit. The generated timing constraints capture what is needed to obey the protocol interface.

To model timing constraints, we use relative timing methodology [49] which is based on the ordering of events. Although not practical, the glitch would have been avoided in Figure 1.1 if there were a relative timing constraint such as "*when IN changes, the INV's output must change before OUT changes.*" This can be interpreted as "the path from *IN* to *OUT* must be slower than the path from *IN* through *INV* to *OUT*." To guarantee that this is always true, a delay exceeding

the amount of delay in the path *IN–INV–OUT* could be added in the path between *IN–OUT*.

The complete set of timing constraint is carefully analyzed to form a generalized timing constraint which becomes stored in the design library. For larger components that belong to the same circuit family, one can use a known starter set of timing constraints from the generalized constraints to find the missing timing constraints. Once the design library is complete, designers can use the components without redoing timing verification.

1.4 Contributions

This work is relevant for designing self-timed circuits that use timing constraints. In practice, very few self-timed circuits can work without timing constraint [16, 22].

We developed *ARCtimer*, a framework of formal modeling and verification methods for generating and verifying timing constraints for handshake components with bounded-bundled-data protocol. Through this framework, we can uncover what is needed to make the component's gate-level circuit follow the component's handshake protocol.

Our focus was to ensure that not only the circuit obeys the handshake protocol but also its timing constraints and static timing analysis code are sufficiently general for use in a design library. To achieve this, we analyzed the design patterns and generalized the timing constraints into timing constraint patterns which are also more intuitive.

We build up a shared understanding of what a framework like ARCtimer entails, and helps readers understand the tradeoffs and decisions, and identify essential decision points, the choices one can make, what we and others chose, and why.

We have exchangeable solutions in three areas: STA loops kept intact, DI protocol specifications, failure analysis heuristics to derive timing constraints.

We explain how to model each component, how to model the protocol into properties in a model checker. While doing so, we found that there is a conflict between the design paradigm of semimodularity used since the early days and the new paradigm of relative timing introduced recently to make self-timed circuits fast and efficient. We show this conflict by redefining semimodularity in a way that fits rather than fight relative timing. This work was published in [32].

The model checker we use is a general purpose tool which is open to public access. The use of a specialized tool for timing verification for self-timed circuits carry hidden assumptions. Custom tools are also less flexible in case a user wants to add new features.

Following is a list of contributions from this thesis:

1. Upfront modeling on a component by component basis to reduce the validation effort required to:

   (a) reimplement components in different technologies, by using the notion of patterns seen in Chapter 5, Section 5.2.3.3 and Figure 5.20

   (b) assemble components into systems as shown in Chapter 5, Section 5.1.2 and Figure 5.3, and

   (c) guarantee system-level timing closure by using ARCtimer introduced in Chapter 5, Section 5.2 and Figure 5.7.

2. Modeling of data and bounded-bundled-data timing constraints that permit the control signals to lead or lag behind data signals to optimize system timing. This is shown in Chapter 6, Figure 6.2.

3. New semimodular model [32], a key property present in most self-timed modeling tools such as [30,47,63]. The enhanced definition of semimodularity can be found in Chapter 3, Section 3.7.2 and Definition 3.7.2.

I have published the following papers:

1. Journals

   (a) H. Park, A. He, M. Roncken, and X. Song. *Semi-Modular delay model revisited in context of relative timing.* IET Electronics Letters, 51(4):332–334, 2015 [32]

   (b) H. Park, A. He, M. Roncken, X. Song, and I. Sutherland. *Modular timing constraints for delay-insensitive systems.* JCST, Springer, accepted 2015 [33]

2. Conferences and Poster sessions

   (a) M. Roncken, S. Metta Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland. *Naturalized communication and testing.* In Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on, pages 77–84, May 2015 [39]

   (b) H. Park, A. He, M. Roncken, X. Song. *Verifying Timing Constraints for Delay-Insensitive Circuits.* Poster presentation. In Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium, May 2015.

   (c) M. Faust, H.Chung, H. Park, J. Rodriguez. *Introducing hardware emulation in the ECE curriculum.* In IEEE International Conference on Microelectronic Systems Education (MSE '11), pages 39–40, June 2011.

1.5 Organization of the Dissertation

The rest of the dissertation is organized in seven chapters.

Chapter 2 reviews related work on self-timed circuit verification.

Chapter 3 describes the basic fundamentals of asynchronous circuits, delay models, circuit and environment model, and relative timing methodology. An enhanced definition of semimodularity is presented with an example of a C-element.

Chapter 4 describes how modeling and verification is done in a general purpose model checker using a C-element as an example.

Chapter 5 presents ARCtimer, our timing verification framework for generating and verifying delay-insensitive circuits. A storage element from Click circuit family is used as an example. We show how relative timing constraints are derived and modeled, and then generalized into timing patterns.

Chapter 6 adds datapath to the storage example and show how data is modeled, and what type of timing constraints are required for the bounded-bundled-data protocol.

Chapter 7 concludes this thesis and addresses possible future work.

2

Related Work

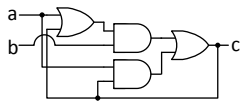
Timing closure for self-timed digital circuits is a problem of a high relevance, because very few circuits, if any, are insensitive to wire and gate delays [16, 22]. Several approaches have been proposed to tackle timing verification.

Kenneth Stevens *et al.* in [50] introduced *Relative Timing* (RT) methodology for synthesis and also for verifying asynchronous circuits that use unbounded delay model. Relative timing constraints the overall delay of two paths such that a specified path is faster than the other. The two paths has a common starting point, which is called Point-Of-Divergence (POD), and has two different ending points, called Point-Of-Convergence (POC_0 and POC_1). The RT constraint shown in Figure 2.1 reads as "*if event* POD *happens, event* POC_0 *must happen before event* POC_1." See Section 3.6.

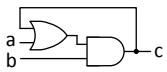
$$\boxed{POD \rightarrow POC_0 \prec POC_1}$$

Figure 2.1: RT constraint is expressed as a path from point-of-divergence (POD) to two different point-of-convergence (POC). From POD, the delay of the path to POC_0 must be less than the path to POC_1.

Relative timing methodology makes timing requirements explicit. Timing requirements of a circuit can be directly added, removed, and optimized using this style. When used for synthesis, depending on the environment, the circuit can be simplified with RT constraints since it reduces concurrency in the implementation. Figure 2.2b from [50] shows an example of a simplified C element with the assumption that the environment always changes input a from high to low before b changes from high to low.



(a) Speed-independent C-element



(b) With assumption $a- \prec b-$

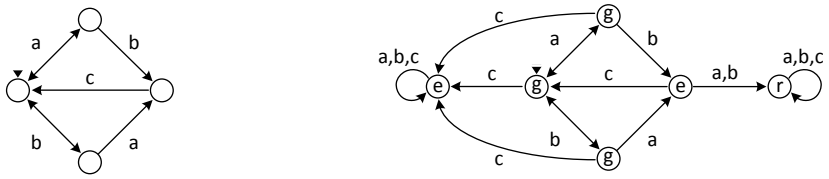
Figure 2.2: Simplification in synthesis. Timing assumptions of the environment can lead to a simpler circuit as shown in [50].

For verification, they use a custom verification tool called Analyze which use trace semantics and *calculus for communicating system* (CCS) based *logic conformance* relation [48]. They model and verify that the timing-constrained circuit meets the protocol. The application of timing constraints can be aggressive or conservative depending on the application.

Yang Xu [62, 63] builds upon [18] and [50]. He created a tool called ARTIST which automatically generates timing constraints based on bisimulation formalism. Error traces or action sequences are evaluated and a RT constraint is generated and added to the circuit implementation. The tool iterates through evaluation and generation of RT constraints until the circuit implementation conforms to the specification. However, automation typically pushes constraints to become too fine grain to provide intuition. Also, the constraints are technology dependent.

Krishnaji Desai *et al.* in [9] models the circuit and RT constraints in NuSMV model checker to check for correctness. They demonstrate the growth of the system size as pipeline stages become deeper. However, progress and choice equivalence properties are absent from the NuSMV based timing verification work.

Radu Negulescu *et al.* in [30, 31] in Process Spaces and FIREMAPS built a system very much like Ken's. The difference is that they use complete path of events as constraints instead of start and end points of a path. Their research are examples that use the theory of *Delay-Insensitive Algebra* for both modeling and verification task.



(a) Original state graph **(b)** Process representation

Figure 2.3: Constructing a process representation using Delay-Insensitive Algebra as shown in [30]. The state machine needs to be a complete automaton so in addition to the original state graph, they add *reject* state (ⓡ), *error* state (ⓔ), and *goal* states (ⓖ). An error from the environment leads to a *reject* state where it can't escape while an error from the circuit leads to an *error* state which it can't escape.

We use a similar approach for describing the specification of the protocol when expanding the state graph. We have two types of error states, error from the environment and error from the circuit. For the timing constraints, they use a term called *Chain Constraint* which is a path based constraint. The path is expressed with all the intermediate events on the path, listed like a chain, which the delays to the early and late events are calculated. Chain constraints are much easier to translate into STA code than relative timing constraints. However, chain constraints are also much harder to model in a model checker than relative timing constraints. We add checkpoints in our path of relative timing constraints to perform STA.

Prasad Joshi [15] verified single track bi-directional wires for the GasP component, and worked on handling loops for STA tools by breaking timing loops. Since conventional STA flow does not support bi-directional wires, he split these wires to verify GasP circuit family. He applied RT constraints on the control logic to satisfy the specification.

Yoneda *et al.* in [64] uses metric timing. Using absolute delay reduces complexity in state space, and they suggest that un-timed circuits may introduce

unrealistic failure traces complicating the verification process. They model each circuit element with a time Petri net. Using min–max delay is a different approach, however, this seems to fit less with self-timed design compared to relative timing. This method also makes the constraints technology dependent.

Khaled Alsayeg *et al.* in [1] created Requirement Analysis Tool (RAT) to formally verify asynchronous circuits using model checking techniques. The tool checks for the correctness of the behavior using *Property Specification Language* (PSL). They describe the circuit using a set of properties expressed in temporal logic, and use a model checker to verify the properties. They start with specifying properties for a general gate model, and then move to abstract level and check the interconnection of each block.

3

Fundamentals and Semi-modularity Revisited

Self-timed circuits, also known as asynchronous circuits, operate on a handshake protocol and communicate through channels with their neighbor modules. Ensuring that each module operates correctly according to the handshake protocol, one can easily build scalable systems from these self-timed components. We build our self-timed designs from circuit components that interact using handshake protocols. The designs that we consider are delay-insensitive as long as every component faithfully follows the handshake protocols. By carefully considering time locally, we can ignore time globally. We use Relative Timing (RT) methodology [9,62] and build upon it to enforce local timing relations as known as RT constraints.

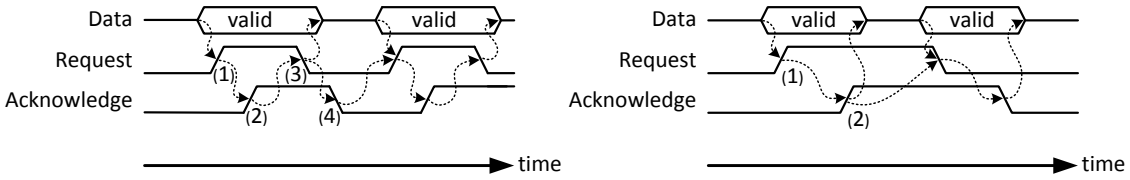
This chapter introduces the basics of communication protocols, graphical representation of circuits, gate and wire models, coloring schemes. We also discuss definitions of semimodularity, and bring in an enhanced definition of semimodularity used in the presence of RT constraints, and show by example of a Muller C element how the enhanced definition is applied.

3.1 Asynchronous Communication Channels

Contrary to synchronous circuits where every action happens on a beat of a clock, asynchronous circuits communicate with neighbors through communication channels with the use of handshakes. This can be thought of as an interaction between two people, one person sending a request as needed, and the other person acknowledging the request at his own pace. In synchronous system, these two

people would only be able to communicate during the tick of a clock, and the clock would keep ticking even when there is no real work going on.

Bundled-data four-phase and *two-phase* handshakes are widely used handshake protocols. Bundled-data refers to data wires being bundled with a separate request and acknowledge wires. The four-phase and two-phase naming comes from the number of transitions required in the request and acknowledge wires to complete a handshake event.



(a) Four-phase RTZ protocol **(b)** Two-phase non-RTZ protocol

Figure 3.1: Timing diagram of bundled data handshake protocols.

Four-phase handshake: (1) The sender places the data and sends a request signal and valid data. (2) The receiver detects this received data, and acknowledges. (3) The sender responds to the acknowledgement by lowering the request signal, and by generating new data. (4) The receiver acknowledges this and lowers the acknowledge signal. [1]

Two-phase handshake: (1) The sender places valid data and sends a request signal. (2) The receiver receives the data and acknowledges.

In digital circuits, the voltage level of a wire is represented with Boolean value 0 or 1, for low and high respectively. A transition on a wire means the voltage level changed from low to high, or high to low. Comparing the four-phase to the two-phase non-return-to-zero handshake protocol, the four-phase protocol has the advantage of simpler circuitry in that it uses level signaling for the control and

[1] Various data validity schemes exist. See Ad Peeters 1996 Ph.D. dissertation [35] for details.

allows each state to be unique, making it easy to determine the initial state of each handshake. Its disadvantage is the extra return-to-zero (RTZ) transitions which take extra time and power. The two-phase handshake protocol could lead to a faster circuit since there are fewer signal transitions, but the circuit is typically more complex.

3.2 Circuit Class

Asynchronous circuits can be classified by their delay model. In this thesis, unbounded delay means a positive but possibly infinite delay, while bounded delay means a positive but finite delay. Muller's *Speed-Independent* (SI) circuits [29] with an example in Figure 3.2a use an unbounded delay model on the gates. Known as *iso-chronic forks*, either the wires are assumed to have negligible delay, or the wire delays are lumped into the gate which it connects to. In result, when a gate's output changes, all the connected gates immediately sees the change.

Delay-Insensitive (DI) circuits which are the most robust of the three classes do not make any delay assumptions and operate correctly under arbitrary gate and wire delays. This is shown in Figure 3.2b where each gate and wire has its own delay.

The delay model of *Quasi-Delay-Insensitive* (QDI) circuits differ from the SI model in that different branches of a forked wire may have different delays. This means that certain forks are iso-chronic.

(a) SI **(b)** DI

Figure 3.2: Circuit classification depending on the delay model. Square blocks represent gate delay and round blocks represent wire delay.

Very few circuits are DI or even QDI or SI. In general, relative timing constraints are needed on top of these circuits to enable the user to repair the delays to make it behave according to the protocol.

We model our circuits for higher-level delay-insensitive protocol applications where the correctness is independent of gate and wire delays in the sense that we can repair any given gate and wire delay setting to obtain correctness of the application.

3.3 Graphical Representation of Asynchronous Circuits

Asynchronous circuits must be free of hazards, so every gate or wire transition counts and carries a meaning. To express the possible behaviors of the circuit, we use a finite state machine with interleaving semantics, where each node is a unique vector of logic wire levels. A module's communication protocol describes the behavior at a more abstract level and only sees the external inputs and outputs.

Such state machines can be represented as a directed graph using a 4 tuple $G =< S, s_0, E, T >$ as in Figure 3.3b:

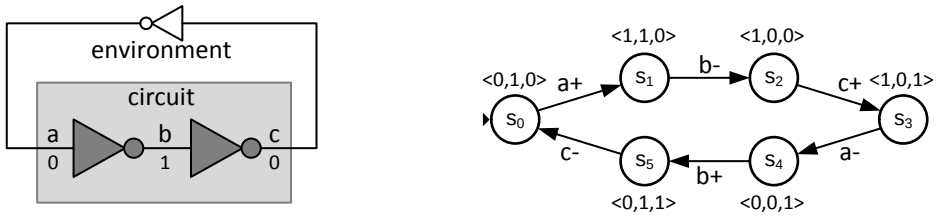
- S: Finite set of nodes represented with circles
- s_0: A set of initial node $\in S$

- E: Event, which is a wire transition from low to high, high to low, or simply any transition
- T: $\{S \times E \rightarrow S\}$: Transition is labeled with an event, leading to a next state. We use interleaving model where only single event happens in a transition

A wire *transition* indicates a change in the logic level of the wire. We use the following notation for transitions:

- $+$: low-to-high transition
- $-$: high-to-low transition
- $\pm$: either low-to-high or high-to-low transition

Let's now look at the system shown in Figure 3.3a. The system part called "circuit" has two inverters connected in series and the system part called "environment" has one inverter. The wires are initialized as $a = 0$, $b = 1$, $c = 0$. After the environment circuit changes a from 0 to 1, it produces the next input only after it sees an output transition on output of c the circuit. A state is defined as $< a, b, c >$, and this system can be modeled with 6 states total if we distinguish up and down transitions. From the initial state $s_0 =< 0, 1, 0 >$, both inverters are stable (colored gray). The only possible action is for the environment to issue $a+$, and go to state $s_1 =< 1, 1, 0 >$. Note that there are no states with $< 0, 0, 0 >$ or $< 1, 1, 1 >$ since these states are not possible in this environment.



(a) Two inverters with environment **(b)** State graph, $s =< a, b, c >$

Figure 3.3: Example of two inverters in series and its state graph.

3.4 Gate and Wire Model

A gate is represented by a Boolean function. We use the theory of Logical Effort [56]. As a result, our circuits come with an "analog health" waiver: their signal rise and fall times are sufficiently good to skip analog circuit analysis.

A logic gate is modeled as a triple $< I, O, F >$ where:

- I is a set of input signals
- O is an output signal
- F is a set of the circuit functions, where after an unbounded delay, O receives the value of F

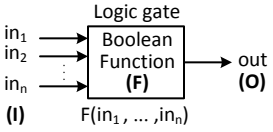


Figure 3.4: Representation of a logic gate.

A *stable* gate means $F(I)$ and O have the same value, and that there is no transition scheduled on the output. We color stable gates gray.

$$out = F(in_1, ..., in_n) \qquad \text{(stable)}$$

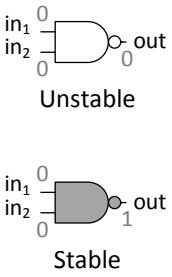
When $F(I)$ has a different value from O, O may change to the value of $F(I)$. We color unstable gates white.

$$out \neq F(in_1, ..., in_n) \qquad \text{(unstable)}$$

Let's look at an example of a two-input NAND gate shown in Figure 3.5. The function of the gate is expressed as $F(in_1, in_2) = \neg(in_1 \wedge in_2)$. When $in_1 = 0$ and $in_2 = 0$, the function $F(in_1, in_2)$ is $\neg(0 \wedge 0)$. In this state, if the current output *out* is 0, this gate is *unstable*. If the current output *out* is 1, then this gate is *stable*. This is shown in Figure 3.5b. The coloring scheme of *stable* and *unstable* will be used throughout this thesis only where necessary.

in_1	in_2	$F(in_1, in_2)$	*out*	Stable
0	0	$\neg(0 \wedge 0)$	0	False
0	0	$\neg(0 \wedge 0)$	1	True
0	1	$\neg(0 \wedge 1)$	0	False
0	1	$\neg(0 \wedge 1)$	1	True
1	0	$\neg(1 \wedge 0)$	0	False
1	0	$\neg(1 \wedge 0)$	1	True
1	1	$\neg(1 \wedge 1)$	0	True
1	1	$\neg(1 \wedge 1)$	1	False

(a) 2-input NAND gate



(b) Example of unstable and stable

Figure 3.5: 2-input NAND gate truth table showing all 8 states, and an example of unstable and stable gates.

As shown in Figure 3.6, an *unstable* gate that is given enough time will eventually become *stable* by changing its output. However, it is also possible that the inputs changes quickly enough that the gate no longer wants to change the output, as it considers itself *stable*. Such example is possible in an inertial delay model.

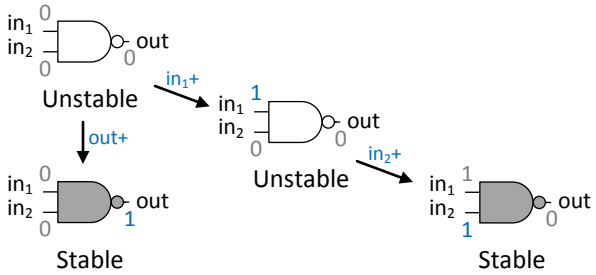


Figure 3.6: Unstable gate may become stable by changing the output (left path), or by changing the inputs (right path).

An inertial delay model can be thought of as a logic gate model with inertia. Inputs have to hold long enough to propagate to the output. Short pulses may be ignored if the pulse is too short.

Figure 3.7 shows all possible actions for a 2-input NAND gate under such an inertial delay model. For example, trace $s_0 \xrightarrow{a+} s_1 \xrightarrow{b+} s_3 \xrightarrow{b-} s_1$ shows that in state s_3, b was retracted too quickly that it wasn't propagated to the output. This trace can also be seen as the gate becoming unstable in state s_3, but because input b changed quickly back to 0, the gate became stable again.

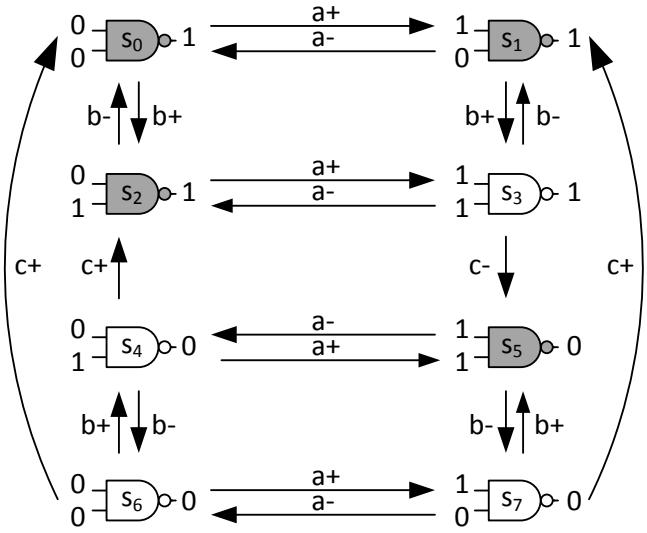


Figure 3.7: Inertial delay model shows that inputs can be withdrawn at any time even before it propagates to the output.

Semimodular delay model introduced by David Muller [29], and brought to the attention of a wider audience through Raymond Miller's 1956 book [27] is widely used for designing hazard-free self-timed circuits by insisting that a digital signal changes occur before being disabled. One might call it the "no change left behind" paradigm. Once an output change is scheduled, it must go through. This puts a strong restriction on the inputs and the surrounding environment. Figure 3.8 shows all possible actions that are allowed in a semimodular delay model for a 2-input NAND gate. Compared to the inertial delay model in Figure 3.7, there are four fewer possible transitions which are prohibited because the pending output change on c from state s_3 and s_7 cannot be canceled. From an unstable state s_3, the only possible action is to change the output by doing $c-$ and going to state s_5. From another unstable state s_7, as long as the input change doesn't make the gate stable, it is allowed to take that action, which is why going to state s_6 by $a-$ is legal.

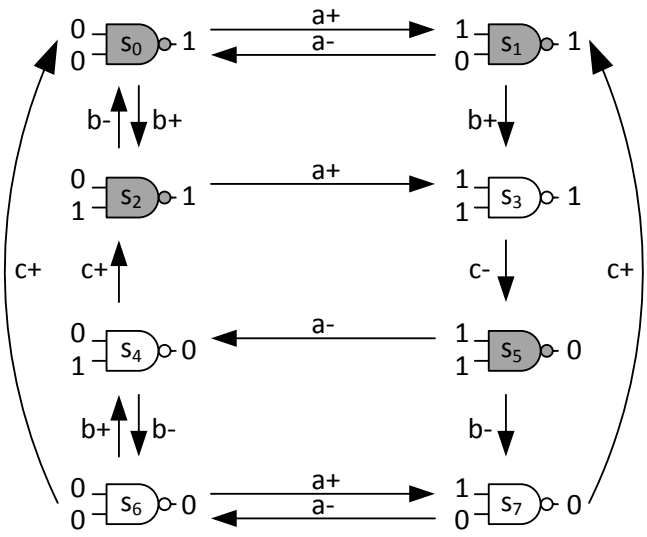


Figure 3.8: Semimodular delay model shows that when an output change is scheduled, it cannot be canceled by changing an input. This is seen in state s_3 where it cannot do $a-$ or $b-$. In state s_4, action $a+$ is not possible, and in state s_7, action $b+$ is not possible.

Transport delay model which is also used with inertial delay model in hardware description language during functional simulation assumes that no matter how short a pulse is, it is scheduled and propagates to the output. Transport delay model is used for functional simulation and timing analysis, and it is typically used with fixed delay range. Semimodular model is a logic model used for logic simulation and verification analysis of arbitrary delay range, but other than that they express the same thing but used for different purpose.

Wire delays are making a bigger presence on the circuit with advances in process technology. The wire delays on different paths of a fork can't be ignored. To model the difference in wire delay on forks, buffer gates are added in each wire branch as done in [46].

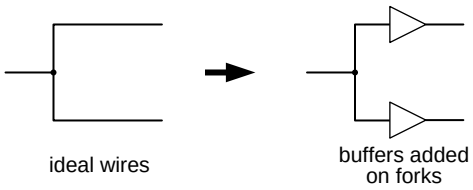


Figure 3.9: Buffers are added to model delays in the branches of forked wires

3.5 Environment Model

Every component operates in some environment which provides the inputs. Overcomplicating the environment leads to unnecessary computation, while oversimplifying will not accurately model the real environment. There are basically two modes of operation of the environment, *fundamental mode* pioneered by [13], and *input-output mode* pioneered by [29]. Both modes assume that the circuit starts in a stable state. *Fundamental mode* allows the environment to change one input, and waits until the entire circuit is stable. Only then can it change the next input. *Input-output mode* allows to change the inputs, and when the circuit

responds by producing any output, the next inputs can be changed again. This means there could be internal signals that are not yet stable before another input changes.

The environment model we use are similar to input-output mode, where inputs can change as soon as an output is produced. We assume our components operate in an environment where the behavior of the environment is well known, and the inputs follow the interface specification, which is also discussed in Section 3.8.1.

In Figure 3.10, the environment supplies inputs to the circuit, and the circuit responds to those inputs by producing outputs. The dashed lines mark the area the specification cares about – getting the correct sequence of inputs and outputs for the circuit. If the circuit does not behave according to the specification, the circuit can be repaired or reconfigured so that it produces the correct outputs. However, this assumes that the correct inputs were provided according to the specification in the first place. If the inputs were incorrectly provided by the environment, it would be meaningless to verify the circuit.

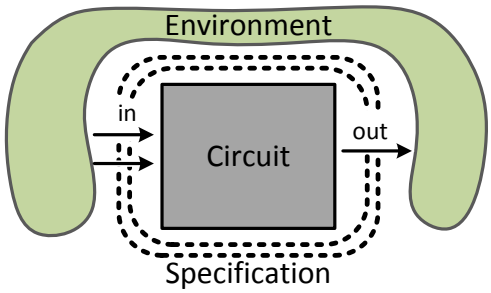


Figure 3.10: Environment provides inputs for the circuit according to the channel protocol specification. The circuit responds to the inputs and produces outputs. From the circuit's perspective, the environment supplies inputs while the circuit produces outputs.

In the model checker that we use, the environment can be modeled as a randomly behaving environment or it can be modeled as part of the implementation. For the randomly behaving environment, there needs to be an invariant to check

only for the inputs that the circuit really cares about and to discard incorrect input events. If the environment is designed as part of the implementation, it should produce inputs according to what the circuit specification expects.

3.6 Relative Timing Methodology

Only a small class of self-timed circuits can work correctly under arbitrary gate and wire delays. *Relative timing* methodology introduced by Kenneth Stevens [50] specifies the relative ordering of two events making all timing requirements explicit. Relative timing constraints specify event orderings that—when obeyed—guarantee correct operation of the circuit. There are various ways to express these constraints. We follow [9, 62] and express them as triples (POD, EARLY, LATE), where:

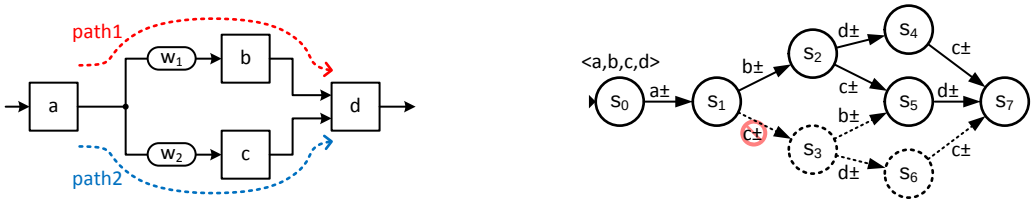
- *POD* is the "point of divergence" event that causes the target events.
- *EARLY* is the target event following POD that must happen before the LATE event.
- *LATE* is the target event following POD that can happen only after the EARLY event.

Each relative timing constraint is expressed in the form of:

$$POD \rightarrow EARLY \prec LATE$$

Enforcing a relative timing constraint on two circuit events, POD is the common point where the circuit paths to where two events begin. The POD tells when the constraints begin to be enforced. In Figure 3.11a, this would be on the output of gate a. The two forked wires from output a have arbitrary delays and go respectively to gate b and c. The two paths through b and c, path1 respectively

path 2, merge at gate d. In case we want to guarantee that path 1 is faster than path 2, we create a relative timing constraint using a as POD, b as EARLY, and c as LATE.



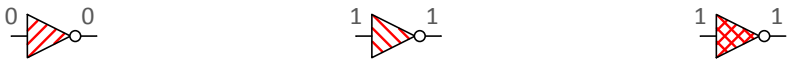
(a) Two paths from a to d **(b)** S_3 and S_6 becomes unreachable

Figure 3.11: Enforcing a relative timing such that path 1 is faster than path 2 is done by placing RT constraint as $rt_1 : a\pm \rightarrow b\pm \prec c\pm$. With this constraint, the state graph on the right shows that s_3 and s_6 are unreachable.

Relative timing constraints are guaranteed by adjusting the delay settings of gates and wires in the late path of the circuit, and validated using static timing analysis. Since timing can directly affect the performance and robustness of the circuits, each constraint can be evaluated, and its application can be aggressive or conservative.

In the context of state transition graphs, the basis of RT methodology is to avoid an illegal state by side-stepping to another legal state as shown in Figure 3.11b. Avoiding one state over another is enforced once POD happens. The EARLY event always happens before the LATE event. In the circuit, this means that the path delay from POD to EARLY is smaller than the path delay from POD to LATE.

When an output transition of a gate is blocked by an RT constraint because it's a LATE event, we indicate the direction (low-to-high or high-to-low) of the blocked transition by using stripes slanted in the direction which is being blocked. Figure 3.12 illustrates an inverter that has a boolean function $out = \neg in$, with one or more constraints blocking the output transition.



(a) *out*+ is blocked **(b)** *out*− is blocked **(c)** *out*± blocked

Figure 3.12: Stripes slanted in the direction of the output transition being blocked. All three gates are unstable, but the output is not allowed to change because it is blocked by one or more relative timing constraints.

3.7 Semimodularity in the context of Relative Timing

Semimodularity is a well-known paradigm for designing hazard-free self-timed digital circuits. Semimodularity requires that a digital signal change—when enabled—must happen before it is disabled – see Section 3.4 and Figure 3.8. Semimodularity played a key role in the early development of computer aided design tools for self-timed systems. Introduced by Raymond Miller [27], it was the starting point for the first generation of self-timed design and analysis tools [19,25,60]. The early tool focus was on generating circuits that—though large and slow—were correct, independent of the gate and wire delays in the design.

Focus shifted to speed and energy efficiency, which was achieved by exchanging delay-*in*sensitivity for extra delay assumptions formulated as relative timing constraints [8, 9, 30, 49, 62]. By adding relative timing constraints, the circuits behave as a delay-insensitive circuit. But the definitions of semimodularity were neither re-examined nor adapted in the context of relative timing. In this section, we will show the need for and present a new definition of semimodularity that is aware of relative timing constraints. The results in this section were published in [32].

3.7.1 Semimodularity—old definition

As explained in Figure 3.6, an unstable gate can become stable by changing either its output or its inputs. Semimodularity allows the change of output but forbids changing the inputs in case this change causes a gate to become stable.

Our execution model for changes is based on finite traces of events, i.e. gate or wire transitions, and on an interleaving semantics that represents parallel events by arbitrary sequential orderings of the events. The resulting single event orderings are also used to model the delays of the events, relative to each other. This is a standard execution model for analyzing self-timed designs. Under this execution model, semimodularity for gates with arbitrary transition delays can be formulated as follows:

Definition 3.7.1. An unstable gate sees no changes until its output changes, i.e.

$$out \neq F(in_1, \cdots, in_n)$$
$$\rightarrow$$
$$\Big[\big\{out' = F(in_1, \cdots, in_n)\big\} \vee (out' = out)\Big] \wedge \big\{F(in'_1, \cdots, in'_n) = F(in_1, \cdots, in_n)\big\}$$

□

In addition to the gate model explained in Section 3.4, the tick symbol($'$) represents the next value of a symbol. Definition 3.7.1 can be read as: "If a gate is unstable, then the output may change while the function of the inputs stay the same." The output may not change in case the output transition is blocked by a timing constraint. Also note that the last part of Definition 3.7.1, $\big\{F(in'_1, \cdots, in'_n) = F(in_1, \cdots, in_n)\big\}$, means that as long as the function of the inputs do not change, inputs may change.

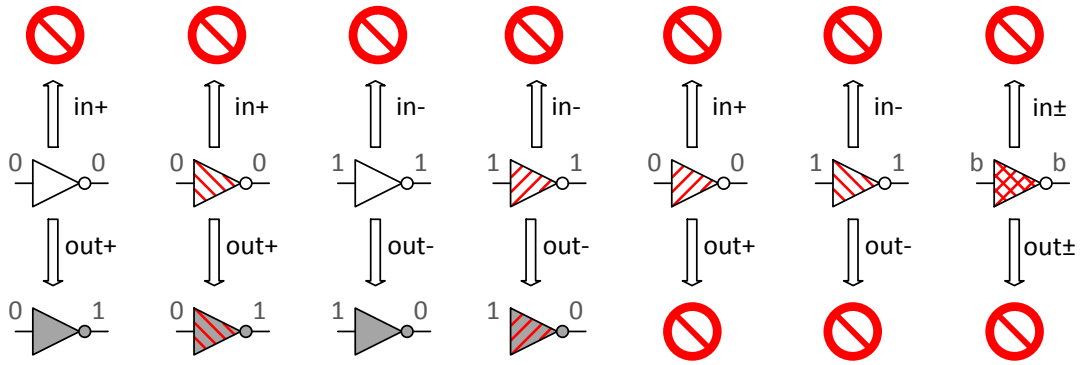


Figure 3.13: Examples of allowed and forbidden transitions for a gate with Boolean function $out = \neg in$. As indicated in Figure 3.5b, we color unstable gates white and stable gates gray. Semimodularity prevents the inputs from changing on all unstable gates. Timing constraints prevents the outputs from changing for the three right gates.

3.7.2 Semimodularity—new definition

A gate output change under the new execution model is enabled if and only if the gate is unstable and the transition that causes the output change is not blocked by relative timing constraints. This loosens up semimodularity so that if the gate's output is blocked by a timing constraint, the function of the inputs can make a transition and make the gate stable. The legal and forbidden execution possibilities for the inputs and outputs are illustrated in Figure 3.14.

In the old definition of semimodularity, an unstable gate can only become stable by changing its output. With relative timing constraints in consideration, we also allow a gate to become stable by changing an input, if and only if the output transition was blocked by one or more relative timing constraints. The legal and forbidden execution possibilities are illustrated in Figure 3.14 for an inverter gate with function $out = \neg(in)$.

As before, semimodularity requires that a digital signal change—when enabled—must happen before it is disabled. In case the output transition is blocked

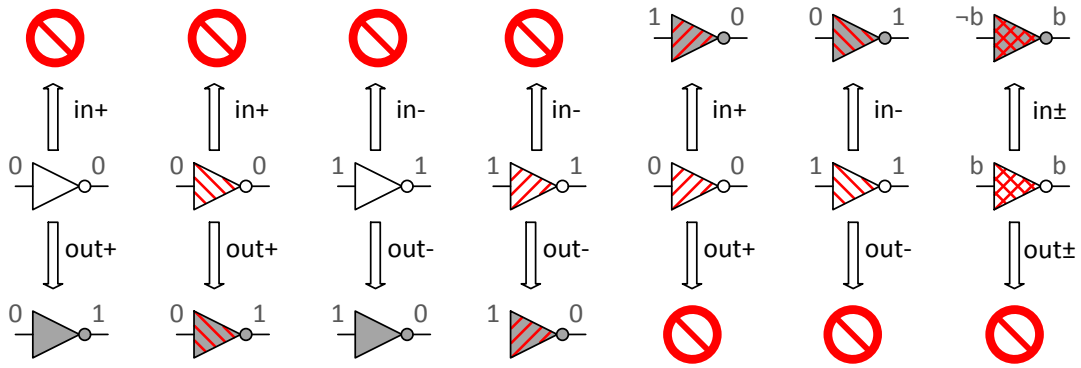


Figure 3.14: Under the new definition of semimodularity, the three right gates illustrate that output transitions are blocked by one or more relative timing constraints. For these three gates, inputs are allowed to change and make the gate become stable.

by one or more relative timing constraints, the inputs may change to make that gate stable. This leads to the following definition of semimodularity in the new execution model with relative timing:

Definition 3.7.2. An *unblocked* unstable gate sees no internal changes until its output changes, i.e.

$$\{out \neq F(in_1, \cdots, in_n)\} \wedge \Big[\{out \wedge \neg block(out-)\} \vee \{\neg out \wedge \neg block(out+)\}\Big]$$
$$\rightarrow$$
$$\Big[\{out' = F(in_1, \cdots, in_n)\} \vee (out' = out)\Big] \wedge \{F(in'_1, \cdots, in'_n) = F(in_1, \cdots, in_n)\}$$

□

Definition 3.7.2 adds an unblocked condition on Definition 3.7.1, and reads as "If a gate is unstable and the output transition is not blocked, then the output may change while the function of the inputs stay the same."

3.8 Example - C element

Muller *C element* is often used in asynchronous circuits as a state holding element. This section uses a C element as an example to go over the specification, an implementation, and how relative timing constraints and the enhanced semimodularity works. The circuit shown in this section is intended to be speed-independent—i.e. wire delays can be ignored.

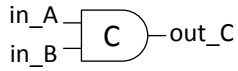


Figure 3.15: Muller C element. When the two inputs have the same value, the output becomes that value. When inputs are different, the output holds its state acting as a state holding element.

3.8.1 Specification

A specification shows the intended circuit behavior. For a C element, when two input values match, the output follows the value of the inputs. The output remains in this state until both the inputs transition to the other state. To express the specification, we use *extended delay insensitive* (XDI) semantics [20].

An XDI description consists of three parts. First is the list of input and output signals. Second is the interface specification which specifies a global sequence of transition events on the channels. Third is the handshake protocol which lists rules that should be obeyed per channel for a correct handshake. Since C element does not have a handshake protocol, the third part is empty from the XDI description and there's only the inputs and outputs signal declaration and the interface specification shown in Figure 3.16a.

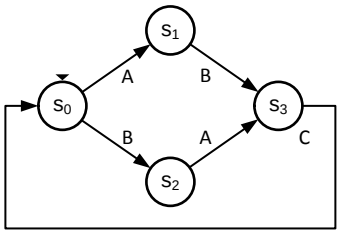
For the C element, the inputs are in_A, in_B, and output is out_C. The interface specification(S) describes that in_A happens and in_B happens, and

then out_C happens. Input events on the interface specification can be moved earlier in the specification sequence, and output events can be moved later in the sequence, as long as they follow the rule specified in the handshake protocol. For the C element, this means that behavior $in\_A?;\ in\_B?;\ out\_C!$ in addition to $in\_B?;\ in\_A?;\ out\_C!$ is also supported.

Figure 3.16b shows the intended behavior in a state transition graph with the initial state, S_0, marked with a small triangle on top of the node. In this graph, A refers to in_A, B to in_B, and C to out_C.

```
1 Inputs and Outputs:
2 I = { in_A, in_B}
3 O = { out_C }
4
5 Interface specification(S)
6 S = in_A? ; in_B?; out_C!; S
```

(a) XDI description



(b) State transition graph

Figure 3.16: Speed-Independent Muller C element specification. Transition A in the state transition graph on the right refers to in_A in XDI description on the left.

3.8.2 Implementation

When it comes to implementation, the environment also needs to be part of the implementation. We modeled a C element using NAND gates with its environment modeled as inverters as shown in Figure 3.17. After the initial inputs from the environment, the next inputs can come only after the output has been produced by the circuit. This environment is not speed-independent, but it represents a general environment where C-to-A path and C-to-B path are modeled separately. The specification checks the transitions on wires A, B, C. We address the circuit gates by their output signal name by using a lower case alphabet. Gates a, b, c are

initially 0, and all gates are stable except for the two inverter gates a and b in the environment.

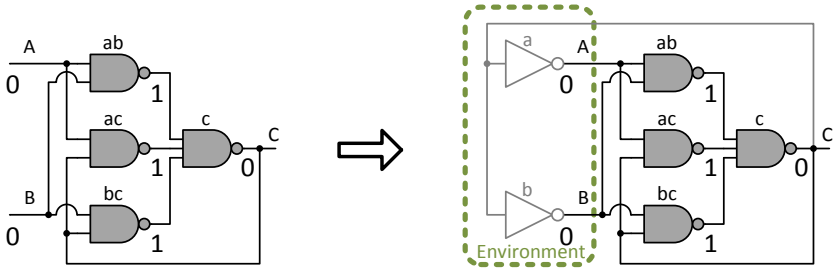


Figure 3.17: A speed-independent C element NAND implementation (left). The environment model represents a general environment where C to A path and C to B path are modeled separately.

From this implementation, a state transition graph including all the internal signals (ab, ac, bc) can be created as shown in Figure 3.18. This is similar to the state transition graph that [50,62] has, but with the difference that we don't enforce semimodularity. The state graph in Figure 3.18 shows parts of the reachable state. In this Figure, dashed edges indicate there are more states that are reachable, but not shown.

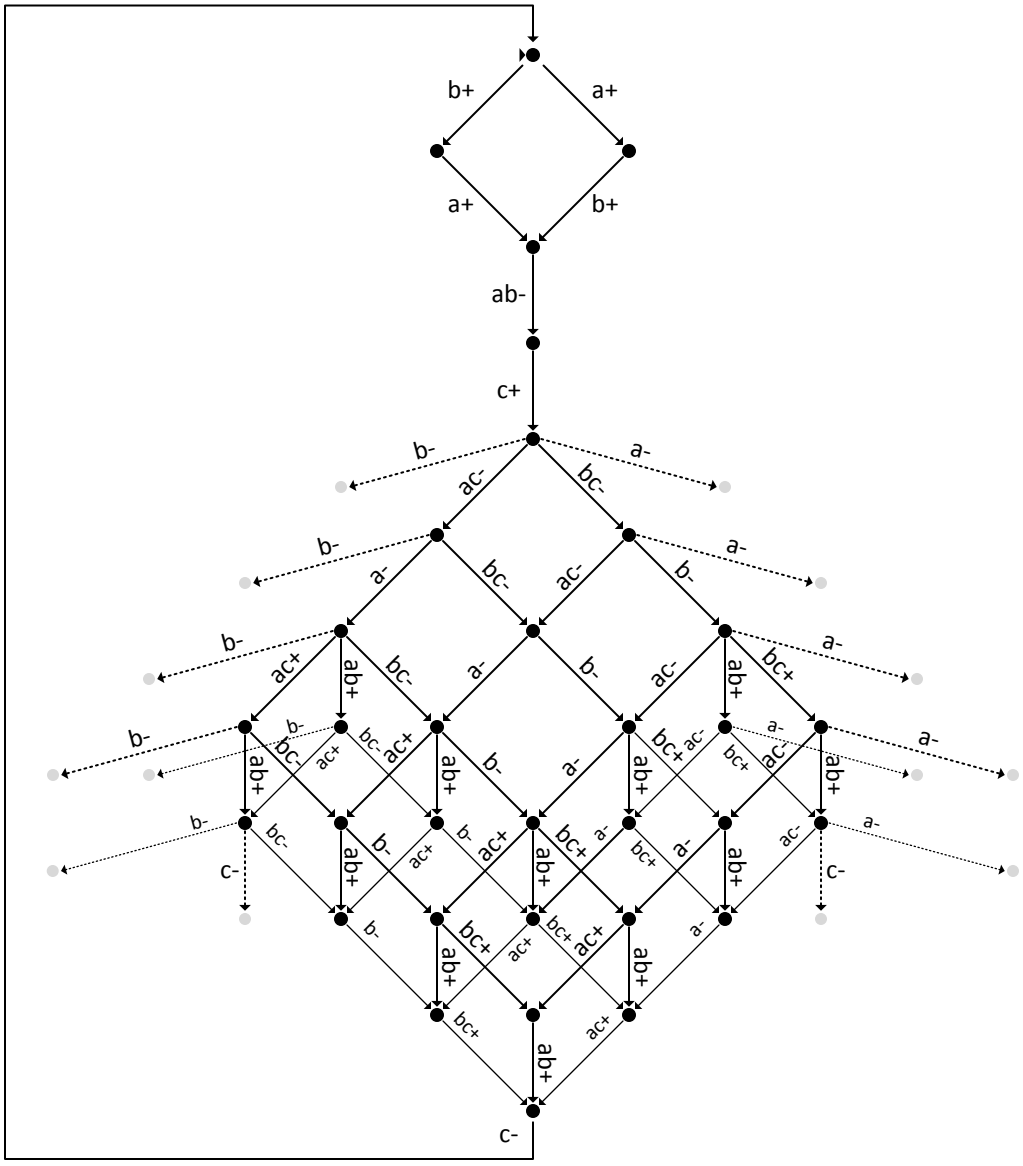


Figure 3.18: State transition graph of C element in NAND implementation as done in [50, 62], but without semimodularity constraint. States are modeled with solid dots. The initial state is marked with a small triangle next to the dot. Dashed edges indicate there are further possible states but not shown.

3.8.3 Applying RT constraints and enhanced semimodularity

The timing constraints derived in [9] for the C element are based on the old definition of semimodularity, using a timing constraint set shown in Figure 3.19a. This solution satisfies the specification, but does not consider the fact that semimodularity and relative timing constraint overlaps with each other. With our enhanced semimodularity definition, we derived four RT constraints shown in Figure 3.19b that give more choices, yet satisfy the specification.

These RT constraints can be generated through formal verification using a model checker. Generating and deriving these RT constraints will be discussed in the next Section of this thesis. For small examples such as the C element, it is possible to visualize how the RT constraints restrict certain paths in a state transition graph.

$c+ \rightarrow ac- < a-$
$c+ \rightarrow bc- < b-$
$c+ \rightarrow ac- < b-$
$c+ \rightarrow bc- < a-$

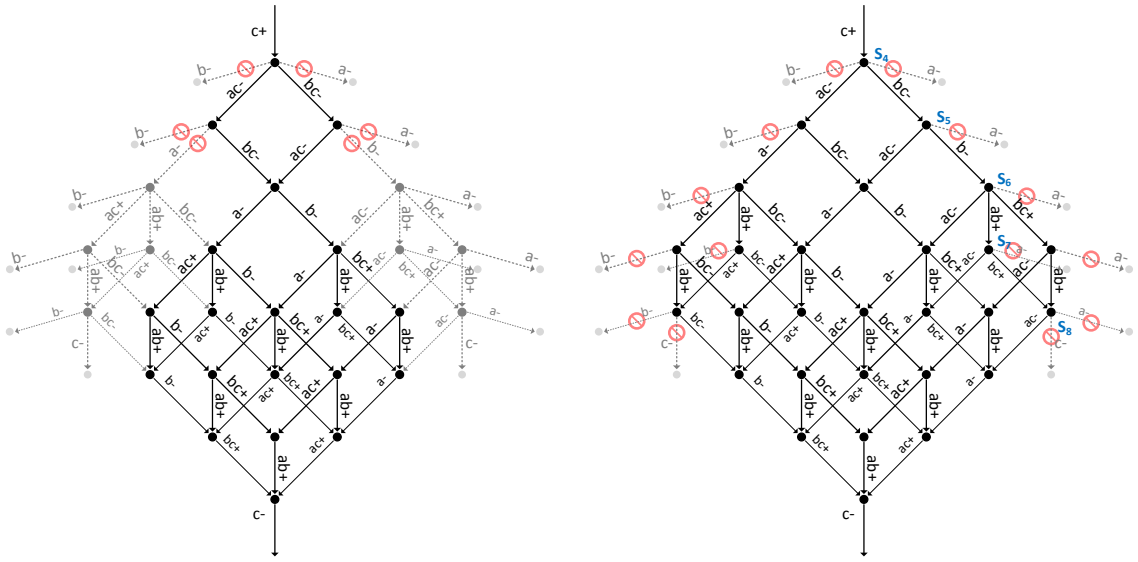
(a) *RT Set 1*

$c+ \rightarrow ac- < a-$
$c+ \rightarrow bc- < b-$
$c+ \rightarrow ac- < c-$
$c+ \rightarrow bc- < c-$

(b) *RT Set 2*

Figure 3.19: Different RT constraints. *RT Set 1* was generated using the old definition of semimodularity, while *RT Set 2* was generated using the new definition.

Looking at Figure 3.20, solid arrow exiting a state indicates possible transitions in that state while a dashed gray arrow indicates transition is no longer possible because the RT constraint set is restricting such transition from that state. Figure 3.20a shows the state transition graph with *RT Set 1*, and Figure 3.20b with *RT Set 2*. This shows that using the new definition of semimodularity, the real circuit behavior can be more flexible than the old semimodularity model allows it to be.



(a) With *RT Set 1* **(b)** With *RT Set 2*

Figure 3.20: Snapshot of Figure 3.18 showing the differences in reachable states with different timing constraint sets applied. Dashed gray lines are not reachable.

Next, let's look at the internal values of an event trace when using *RT Set 2* from Figure 3.19b and the state transition graph in Figure 3.20b. Figure 3.21 shows the states and event trace leading up to what would have been a semimodularity conflict under the old definition, but continues fine under the new definition:

$$S_0 \xrightarrow{a+} \xrightarrow{b+} S_2 \xrightarrow{ab-} S_3 \xrightarrow{c+} S_4 \xrightarrow{bc-} S_5 \xrightarrow{b-} S_6 \xrightarrow{ab+} S_7 \xrightarrow{bc+} S_8$$

The transitions from initial state S_0 to state S_8 are allowed under both the old and new definitions of semimodularity. All four relative timing constraints kick in at state S_4, after POD event $c+$, and start blocking the LATE events, $a-$ and $b-$ and $c-$, until the constraints are released by the corresponding EARLY events, $ac-$ or $bc-$ or both. Event $b-$ is released first, in state S_5. The other two remain blocked up to and including state S_8.

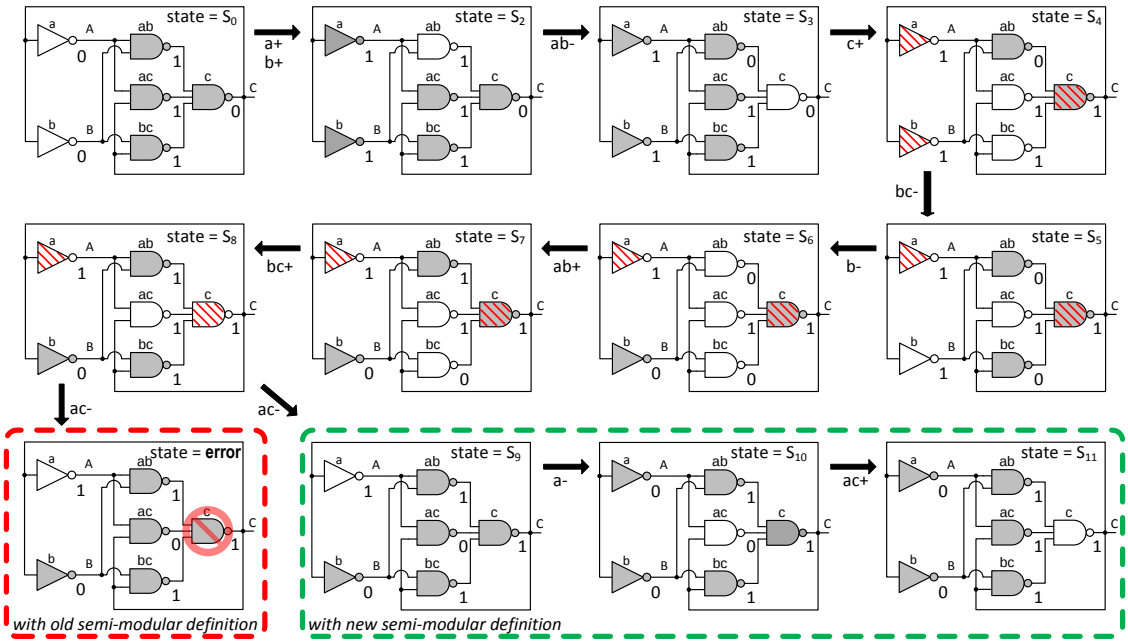


Figure 3.21: Event trace in the new execution model with *RT Set 2*. Events from S_0 to state S_8 are allowed under the old and new definitions of semimodularity. With the old definition, the trace cannot exit S_8 without violating relative timing or semimodularity. With the new definition, there is no semimodularity violation and the trace proceeds correctly from state S_8 to S_{11}.

Under the old definition, the trace stops at S_8—execution cannot exit S_8, because all possible exits would violate either a relative timing constraint or semimodularity. The forbidden exit causing the semimodularity conflict is shown in the bottom left dash lined box in the picture: event $ac-$ causes unstable gate c to see its boolean input function $\neg(ab \wedge ac \wedge bc)$ change from 1 to 0, before its output changes, i.e. before $c-$.

The new definition does not detect any semimodularity conflict: $c-$ is not enabled in S_8, because it is blocked by a relative timing constraint, as indicated by the diagonal stripes (\) for gate c. Under the new definition, execution proceeds correctly from S_8 to state S_{11}, and from there back to the initial state.

So, the execution model with Definition 3.7.1 (old semimodularity) finds the

circuit with the four relative timing constraints incorrect. With Definition 3.7.2 (enhanced semimodularity), it finds that the combination behaves as specified. In other words, the four constraints from *RT Set 2* are falsely rejected under the old definition, Definition 3.7.1, and accepted under the new one, Definition 3.7.2.

4

Modeling

A model checker checks the specification written in properties against a model of a system. We start by building a model of the system that we would like to verify. Once the model is built, the model checker checks if it satisfies the the properties. The model checker automatically performs an exhaustive check. If a property turns out to be false, a counterexample trace is generated. In case of a timing verification, the user will inspect and analyze the counterexample trace to pinpoint the source of the error and correct the problem by adding timing constraints. This process is repeated until either all properties return true, or the system faces a state space explosion problem.

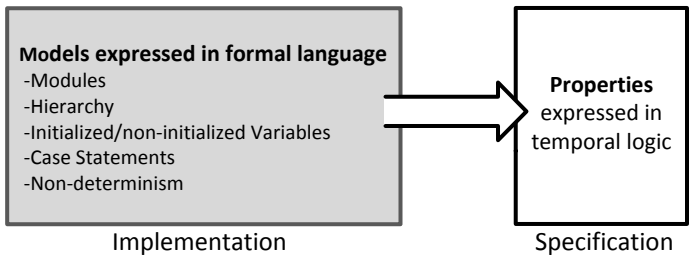


Figure 4.1: Model checker overview. Model of the system is created as an implementation. Properties expressed in temporal logic describes the specification. The model checker goes through the implementation and checks whether the properties hold true or not. The direction of the block arrow indicates that some information is passed on to the Specification. This can be thought of as the Specification observing what the Implementation is doing.

What model checker should one use for this task? The two basic choices are a general-purpose model checker that is widely used or a model checker customized to fit the self-timed computation theory of one's choice. *Analyze* and *Artist* in [52,62] are examples that use customized model checkers with a trace semantics and a

CCS based logic conformance relation. They model and verify that the timing-constrained circuit meets the protocol. *Process Spaces* and *FIREMAPS* in [30,31] are examples that use the theory of Delay-Insensitive Algebra for both the modeling and the model verification task.

A major advantage of a custom model checker is that the theory for the underlying conformance relation and algebraic model is already built into the model checker. On the other hand, a custom model checker tends to have few and highly specialized users and few test examples, and may be flawed by various subtle bugs that make it hard to use for new examples. A major advantage of a widely used open-source general-purpose model checker is that it has many users and many diverse test examples, and so its bugs tend to be discovered and repaired. For this reason, we used NuSMV [6] derived from CMU SMV [24] to verify the correctness of our design.

4.1 Modeling the Implementation

An implementation of a system is composed of a circuit, an environment, and timing constraints as shown in Figure 4.2. Each component generates an event which refers to a transition on a gate or wire. Remember that a signal transition is a signal change from low to high or high to low, as explained in Chapter 3. Each component may monitor and respond to each others events. In the initial stage of modeling the implementation, timing constraints may not be known, so we start with the circuit and the environment model. Note that it is possible to start with a known "starter-set" of timing constraints such as the trivial ones or from past experience within the same circuit family. This will be later shown in the following chapters.

The circuit performs the main functionality while the environment supplies inputs, according to the specification. If the circuit doesn't behave according to the specification, timing constraints are added to avoid the wrong behaviors.

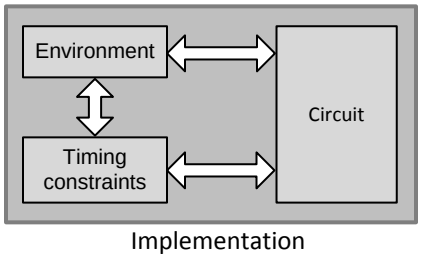


Figure 4.2: Implementation is a parallel composition of circuit, environment, and timing constraints. The arrows indicate that components may monitor and respond to each others events.

4.1.1 Circuit

We code the behavior of a combinational logic gate in a generic MODULE, called *cgate*, with formal parameters *set* and *init_val* and an ASSIGN block to set and initialize output *val* as shown in Figure 4.3b.

(a) Model of a combinational logic gate

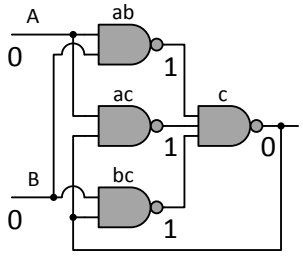
```
1  MODULE cgate(set, init_val)
2     --set         : gate function, also known as set function of the gate
3     --init_val    : initial value for output
4     --val         : gate output
5     VAR
6        val : boolean;
7     ASSIGN
8        init(val) := init_val;
9        next(val) := case
10          set = val  : val; --if set and val are same, then no change required
11          TRUE: set;        --otherwise, change the output to set
12       esac;
13    FAIRNESS running
```

(b) NuSMV code for cgate module

Figure 4.3: *MODULE cgate* models a generic combinational logic gate. Initial value of the output *val* comes from *init_val* when instantiating this module. Depending on the case statement, the *set* function may get copied to the output as shown in line 11.

The parameter *set* is a function with a Boolean result. For example, the set-function for a 2-input AND gate which has in_1, in_2 as inputs are represented as $set = (in_1 \ \& \ in_2)$. The "&" symbol in NuSMV is a conjunction($\wedge$) operator. Line 8 of Figure 4.3b assigns the initial value of the output from *init_ val*, while the case statement in line 9-12 determines whether or not the set function gets copied over to the output. The case statement here simply states that when the current output *val* is different from the *set* value, the *set* value gets copied over to the output *val*. The case statement in its current form is trivial and could have used a single assignment $next(val) := set$, but will become more complex as we add timing constraints and allow stuttering on the output of the gates.

The modules in NuSMV can behave synchronously or asynchronously. The circuit modules are typically instantiated with the keyword *process* as shown in line 7–11 in Figure 4.4b to be instantiated as an asynchronous process [6] and executed by interleaving its operation with those of other *process* instances. The selection of processes are non-deterministic and each process has a special Boolean variable called *running*. This special Boolean variable becomes TRUE when the process is selected, and only one *running* may be TRUE at any time. This causes interleaving of different process operations. For fair interleaving, *FAIRNESS running* is added in each module that instantiates these processes shown in line 12 in Figure 4.4b; this prevents each process from never being selected. The connections between modules are made through shared variables, as shown in Figure 4.4b.



(a) Speed-independent C element implemented with NAND gates. Copy of Figure 3.17.

```
1  MODULE main
2     VAR
3        --inputs can randomly change representing a free environment
4        A: boolean;
5        B: boolean;
6        --two input nand gate
7        ab: process cgate (!(A & B), TRUE);
8        ac: process cgate (!(A & c), TRUE);
9        bc: process cgate (!(B & c), TRUE);
10       --three input nand gate
11       c: process cgate (!(ab.val & ac.val & bc.val), FALSE);
12    FAIRNESS running
```

(b) NuSMV code for C element main module

Figure 4.4: Composition of cgate instances for NAND gate implementation of a C element. The gates are instantiated in the module main. The connections are made by sharing each other's variables, for instance cgate instance *c*.

While modeling the circuit, we also model semimodularity which helps with finding problems related to protocol failures. It's not always necessary to keep semimodularity, but acts as a helper property. Protocol properties are discussed in Section 4.3 and Figure 4.19.

4.1.2 Environment

An environment is what surrounds the circuit and provides inputs while the circuit produces the outputs. Figure 4.5 shows the circuit with a randomly behaving environment. Compared to Figure 4.4a, there are two additional buffer gates, *a* and *b*, instantiated as *process cgate* modules on the inputs of the circuit for interleaving.

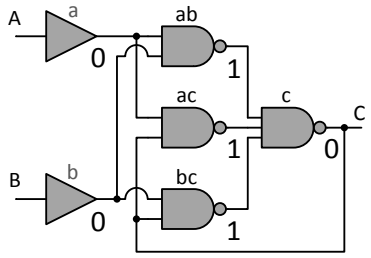
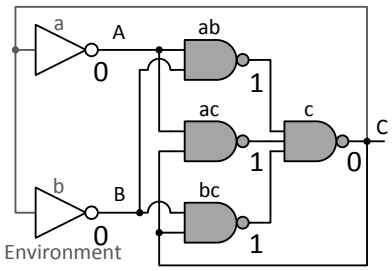


Figure 4.5: C element with randomly behaving environment. The buffers enforce that input events are interleaved.

Since we are only interested in the inputs that are provided to the circuit according to the circuit's specification, one way to filter the environment is to use invariants to ignore the unwanted inputs. Another approach, which we use in our model, is to model the environment as part of the implementation, which provides the inputs that the circuit expects according to the channel part of the specification. We define separate sub environment for each channel. Per channel, the environment responds to the handshake events generated on the channel as output by the circuit. The environment responds by generating the next handshake event on the channel as input to the circuit. The environment response is *lazy* but the response is correct as long as the circuit obeys its part of the handshake protocol on the channel. Components that operate on handshake protocols are also modeled this way since we know what kind of environment the modules are expected to be used in. For the C element, we add inverter gates for input A and B which come from output c as shown in Figure 4.6a. After the initial event from the environment, this allows the inputs to change only after the output changes.



(a) NAND implementation of C element with an environment that has a causal relation between c-a and c-b. New inputs arrive only after the output changes.

```
1  MODULE main
2     VAR
3        --inverters to model an environment
4        a: process cgate (!c.val, FALSE);
5        b: process cgate (!c.val, FALSE);
6        --two input nand gates
7        ab: process cgate (!(a.val & b.val), TRUE);
8        ac: process cgate (!(a.val & c.val), TRUE);
9        bc: process cgate (!(b.val & c.val), TRUE);
10       --three input nand gate
11       c: process cgate (!(ab.val & ac.val & bc.val), FALSE);
12    FAIRNESS running
```

(b) NuSMV code for C element that has environment and circuit

Figure 4.6: Composition of cgate modules for C element. Each module is assigned as a process so that only one module is active at the same time. Each inverter gate in the environment has a set function $\neg c.val$, which means it takes the negation of gate c's output as its input. Gate *ab* has a set function of $\neg(a.val \wedge b.val)$.

A more realistic environment may stall A and B as long as it wants. To model a stalling environment, we add a formal parameter called *lazy* in *module cgate* and make it TRUE for all environment gates. When the *lazy* variable is TRUE, the cgate is allowed to stall the output indefinitely. This is done by altering the *case* statement in Figure 4.7b line 12 such that the next output *val* can non-deterministically keep the current value of *val* or change it to match the value of *set*.

(a) Module cgate with lazy

```
1  MODULE cgate(set, init_val, lazy)
2     --set         : gate function, also known as set function of the gate
3     --init_val    : initial value for output
4     --lazy        : allows stalling the output
5     --val         : gate output
6     VAR
7        val : boolean;
8     ASSIGN
9        init(val) := init_val;
10       next(val) := case
11          set = val  : val; --if set and val are same, then no change required
12          lazy: {val, set}; --if lazy, output can either change or stall
13          TRUE: set;        --otherwise, change the output to set
14       esac;
15    FAIRNESS running
```

(b) *MODULE cgate* with *lazy*. As environment gates are marked *lazy*, they can stall their outputs indefinitely.

```
1  MODULE main
2     VAR
3        --inverters to model a lazy environment
4        a: process cgate (!c.val, FALSE, TRUE);
5        b: process cgate (!c.val, FALSE, TRUE);
6        --two input nand gates
7        ab: process cgate (!(a.val & b.val), TRUE, FALSE);
8        ac: process cgate (!(a.val & c.val), TRUE, FALSE);
9        bc: process cgate (!(b.val & c.val), TRUE, FALSE);
10       --three input nand gate
11       c: process cgate (!(ab.val & ac.val & bc.val), FALSE);
12    FAIRNESS running
```

(c) NuSMV main module with environment and lazy parameter

Figure 4.7: An environment behaving according to the circuit's specification is modeled as part of the implementation. With the addition of *lazy* parameter, each environment gate is allowed to stall its output as long as it wants.

4.1.3 Modeling RT constraints

A relative timing constraint is modeled with a set-reset type finite state machine as shown in Figure 4.8 which was outlined in [9]. We use a more complex state machine which can handle guard conditions in the later chapters, but for now, a simple version is shown here for easier understanding.

Each relative timing constraint has this state machine and they operate independently in a synchronous manner, meaning that the constraints are always evaluated every time any event happens.

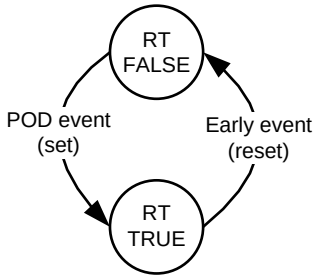


Figure 4.8: RT constraint variable is represented as a state machine. When the *POD* event happens, the RT constraint variable is set to *TRUE*, which is used to block the output of the *Late* event. When the *Early* event happens, the RT variable is *reset* making RT *FALSE* so it no longer blocks the output of the *Late* event. This model assumes that the *POD* and *Early* event are interleaved.

A transition on an output of a gate is referred to as an event. With some exceptions, most relative timing constraints are initialized to *FALSE*, and only become *TRUE* between the POD event and the Early event. When a relative timing constraint is *TRUE*, it prevents the *Late* event until the *Early* event happens and resets the RT value to *FALSE*. When a relative timing constraint is *TRUE*, and the *Early* event has not occurred yet, a constraint placed on the gate producing the *Late* event will disallow the output from changing.

In NuSMV, we model a relative timing constraint in a module called *constraint*, with formal parameters *pod, early,* and *init_val*. The module has an ASSIGN block to initialize output *val*, and a TRANS block to change the output *val* to *TRUE* if *pod* becomes *TRUE*, or to *FALSE* as soon as *early* becomes *TRUE*. The difference between ASSIGN and TRANS is as follows: The ASSIGN statement is used for variable initialization and is executed only when the process is selected. The TRANS statement is about a transition relation in terms of current and next state variable, and is executed at all times.

(a) Module constraint

```
1  MODULE constraint (pod, early, init_val)
2     VAR
3        val : boolean;
4     ASSIGN
5        init(val) := init_val;
6     TRANS
7        next(val) = case
8           !val & !pod   & next(pod)   : TRUE;  --set RT
9           val  & !early & next(early) : FALSE; --reset RT
10          TRUE : val;
11       esac;
```

(b) NuSMV code for the relative timing constraint model in Figure 4.8

Figure 4.9: Relative timing constraints module in NuSMV

To block an output transition due to the transition being a *late* event in an RT constraint, module *cgate* is expanded with two more parameters: *block_HI* and *block_LO* for blocking *low-to-high* and *high-to-low* transitions, respectively. The changes in NuSMV code for the module cgate is shown in Figure 4.10.

```
1  MODULE cgate (set, init_val, lazy, block_HI, block_LO)
2     VAR
3        val : boolean;
4     ASSIGN
5        init(val) := init_val;
6        next(val) := case
7           (block_HI & !val & set) | (block_LO & val & !set) : val;
8           lazy: {val, set};
9           TRUE: set;
10       esac;
11    FAIRNESS running
```

Figure 4.10: *MODULE cgate* with additional code for blocking *late* events.

An overall picture of how the constraints work in relation with POD-early-late in an example is shown in Figure 4.11.

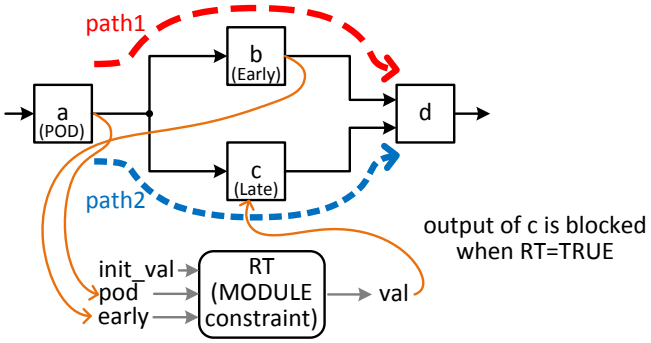


Figure 4.11: Example showing two paths. The RT constraint is $a \rightarrow b \prec c$, which means path1 is shorter than path2. Square boxes *a, b, c, d* are instances of module cgate. The RT constraint module keeps track of *POD* and *Early* event. Upon executing process cgate module instance for gate *c* which produces the *Late* event, the output transition is blocked if RT is TRUE.

4.2 Checking Specification using Properties

A specification is a requirement of a system. The requirements are expressed in properties using temporal logic for the model checker to perform an exhaustive check against the implementation. An overview is shown in Figure 4.12.

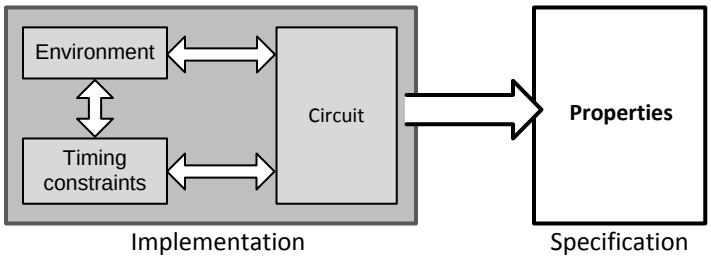


Figure 4.12: Model-checking with implementation and specification. The implementation is composed of circuit, environment, and timing constraints. Specification is expressed in properties and observes the behavior of the implementation checking whether all properties hold true or not.

We write properties using a *computational tree logic* (CTL) language [7], available in the model checker. CTL is a branching time logic, which can be used to describe properties for paths. The syntax is to use one branch identifier

followed by a temporal operator which is again followed by another CTL expression. Figure 4.13 summarizes branch identifiers and temporal operators.

Branch identifier		Temporal operator	
A	For all paths	X	next
		F	in the future
E	There exists a path	G	always
		U	until

Figure 4.13: Basic CTL syntax.

Some basic examples of expressing properties in CTL are shown in Figure 4.14, where p is a CTL expression.

- *AF p*: “wherever you go, p will eventually be true".
- *AG p*: “for all futures p is true".
- *EF p*: “in at least one path, p will eventually be true".
- *EG p*: “in at least one path, p is true forever".

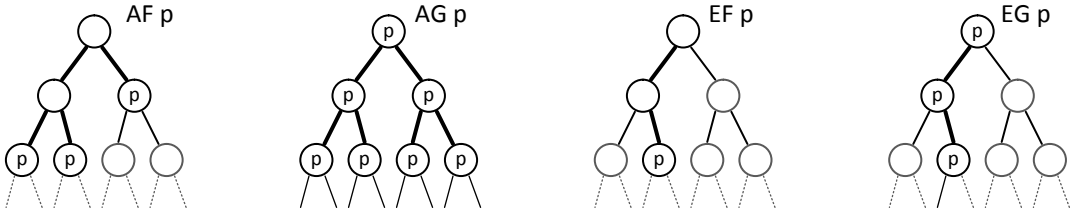
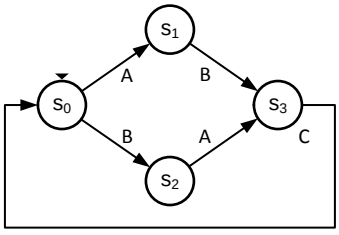


Figure 4.14: Examples of CTL expressing properties from the top of the tree. p is a CTL expression.

We check for *safety* property and *liveness* property. Safety in modeling means that nothing bad happens. In terms of state transition graphs, this means we never get into an error state. Liveness means that something good eventually happens. In terms of state transition graphs, this can be seen as an action being eventually executed with fair choice, making progress. Liveness requires that we add *FAIRNESS running* on the process cgate modules. We enforce liveness by

adding progress properties enforcing that when the output is ready to make a transition it must do so.



(a) State transition graph

$$
\begin{array}{|l|}
\hline
AG\ (\ c.val \rightarrow A\ [\ c.val\ U\ (\ !a.val\ \&\ !b.val\ )\ ]\ ) \\
AG\ (\ !c.val \rightarrow A\ [\ !c.val\ U\ (\ a.val\ \&\ b.val\ )\ ]\ ) \\
AG\ (\ AF\ !c.val\ ) \\
AG\ (\ AF\ \ c.val\ ) \\
\hline
\end{array}
$$

(b) Safety and Liveness properties in a non-*lazy* environment

Figure 4.15: The specification of the C element is expressed with four properties in CTL. Whenever *c.val* is TRUE, it remains TRUE until both inputs *a.val* and *b.val* becomes FALSE. The second property is similar from the first but from symmetry. The third and fourth property is about making progress, where the output *c.val* or *!c.val* can't stay forever. This is assuming the environment is not *lazy*.

Figure 4.15 shows the specification and properties of a C element. In the NuSMV code, the gate name followed by *.val* refers to the output of that gate. For example, a rising transition of A from the state transition graph refer to *!a.val & next(a.val)* in NuSMV. The four properties which check for safety and liveness property shown in Figure 4.15b are added in *MODULE main*.

Similar properties but with a *lazy* environment was expressed in work done in [1] as shown in Figure 4.16.

$$
\begin{array}{|l|}
\hline
AG\ (\ \ c\ \ \rightarrow\ \ (\ A\ [\ \ c\ U\ (\ !a\ \&\ !b\ )\ ]\ )\ |\ AG\ \ c\ ) \\
AG\ (\ !c\ \ \rightarrow\ \ (\ A\ [\ !c\ U\ (\ \ a\ \&\ \ b\ )\ ]\ )\ |\ AG\ !c\ ) \\
AG\ (\ (\ !a\ \&\ !b\ \&\ \ c\ )\ \ \rightarrow\ \ AF\ !c\ ) \\
AG\ (\ (\ \ a\ \&\ \ b\ \&\ !c\ )\ \ \rightarrow\ \ AF\ \ c\ ) \\
\hline
\end{array}
$$

Figure 4.16: Properties shown in [1] are similar to 4.15b but include laziness of the environment.

Writing properties for a small circuit such as a C element is not that complicated. However, more complex circuits that have a larger state space tend to have more properties and the properties tend to become more complex.

We use an alternative method for generating properties directly from *Extended-Delay-Insensitive* (XDI) specification which is explained in the next Section.

4.3 Using XDI Specifications as Monitor and Properties

The XDI specification, written in delay-insensitive algebra, can be used to create a finite state machine which is a collection of all the correct behavior states. On top of the correct states, we add edges with unexpected inputs or outputs leading to an *error* state for every single node. This finite state machine is called *Protocol FSM*. By using Protocol FSM, the burden of writing properties is reduced.

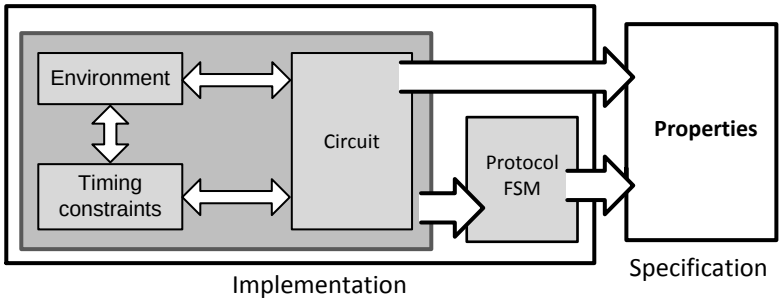


Figure 4.17: Organization for model checker to verify single handshake components. The protocol FSM which comes from an XDI specification also becomes part of the implementation. The protocol FSM monitors the observable actions from the implementation of environment, circuit, and timing constraints. The properties check that the Protocol FSM does not get into an error state and that all legal states from the specification are reachable. One other property that probes the circuit directly shown as an arrow coming from the Circuit to the Properties performs a semimodularity check which could help identifying trouble spots, but isn't a requirement for the behavior to be correct.

The Protocol FSM is also part of the implementation, and operates along the side, monitoring the implementation model of circuit, environment, and timing constraints. All the property checks except semimodularity checks are now performed on the Protocol FSM. Safety properties ensure that error states are not reachable and progress properties ensure that it is possible to make progress

and that all legal transitions are supported as possible implementations. The semimodularity check comes directly from the implementation. Semimodularity checks can be used to help identify problems earlier in the error trace, but may not be essential for a correctly operating implementation. The organization of the new implementation and specification is shown in Figure 4.17.

The C element's XDI specification shown in Figure 3.16 is translated into a Protocol FSM as shown in Figure 4.18. We add all the actions that lead to an error state in this state machine and write properties such as safety, progress, and bisimulation equivalence properties shown in Figure 4.19 line 20-31.

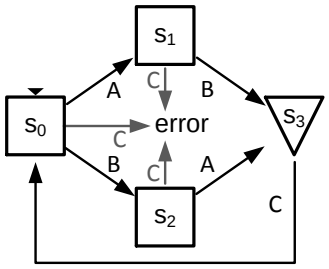


Figure 4.18: Boxes indicate the state is *lazy* and can be stalled. The triangle state indicate that an output is pending so progress must be made. The Protocol FSM monitors the circuit's behavior and changes states accordingly. This monitor assumes that the environment behaves according to the input specification of the circuit. Illegal outputs from each state leads to an error state. Illegal inputs can also be sent to an error state but not shown in this FSM.

```
1  MODULE protocol (in_A, in_B, out_C)
2     VAR
3        state: {s0, s1, s2, s3, error};
4     ASSIGN
5        init(state) := s0;
6     TRANS
7        next(state) = case
8           --legal handshake transitions
9           state = s0 & (in_A  != next(in_A)) : s1;
10          state = s0 & (in_B  != next(in_B)) : s2;
11          state = s1 & (in_B  != next(in_B)) : s3;
12          state = s2 & (in_A  != next(in_A)) : s3;
13          state = s3 & (out_C != next(out_C)): s0;
14          --illegal handshake transitions
15          (state in {s0, s1, s2}) & (out_C != next(out_C)) : error;
16          --otherwise, remain in same state
17          TRUE: state;
18       esac;
19
20    --Safety Property
21    CTLSPEC AG state != error;
22
23    --Progress Property for transient states
24    CTLSPEC AG AF(state != s3) --SPEC !(EF EG (state=s3))
25
26    --All transitions are possible  (Bisimulation equivalence)
27    CTLSPEC AG (state = s0 -> E[state = s0 U state = s1])
28    CTLSPEC AG (state = s0 -> E[state = s0 U state = s2])
29    CTLSPEC AG (state = s1 -> E[state = s1 U state = s3])
30    CTLSPEC AG (state = s2 -> E[state = s2 U state = s3])
31    CTLSPEC AG (state = s3 -> E[state = s3 U state = s0])
```

Figure 4.19: *MODULE protocol* which is generated from an XDI specification. This module monitors the behavior of the implementation and changes states accordingly. When an unexpected output is produced, the state becomes an error state. Safety property on line 21 states that the error state is not reachable, and the progress property on line 24 states that progress must be made when the output of c can change. The *choice equivalence* properties on lines 27–31 spell out the choices of action that must be available to the observed sub-system.

5

ARCtimer

In Chapter 3, the fundamentals and an enhanced version of semimodularity were introduced. In Chapter 4, details of how modeling a C element was shown as an example. This Chapter brings all the previous knowledge into a single framework and shows how to model and verify handshake components. I call this framework *ARCtimer*.

ARCtimer is the timing part of *ARCwelder*, a design compiler we use at the ARC developed by Willem Mallon at Portland State University. ARCwelder is the next version of the design compiler developed by Handshake Solutions [36]. ARCwelder stores descriptions of gate level designs for each component. I add protocol description, RT constraint, and STA code generated from these RT constraints and extend the library of components to a library of verified components. This allows the designer to build a modular and scalable system with timing closure.

Our building blocks are no longer *cgates* as was in Chapter 4, instead the blocks are groups of *cgates* along with other modules which form a handshake component. This Chapter focuses on control of handshake components. On the next Chapter, I will add *bounded bundled data* (BBD) to the verification flow.

ARCtimer is a framework for modeling, generating, verifying, and enforcing timing constraints for individual self-timed handshake components. The constraints guarantee that the component's gate-level circuit implementation obeys the component's handshake protocol specification. Because the handshake

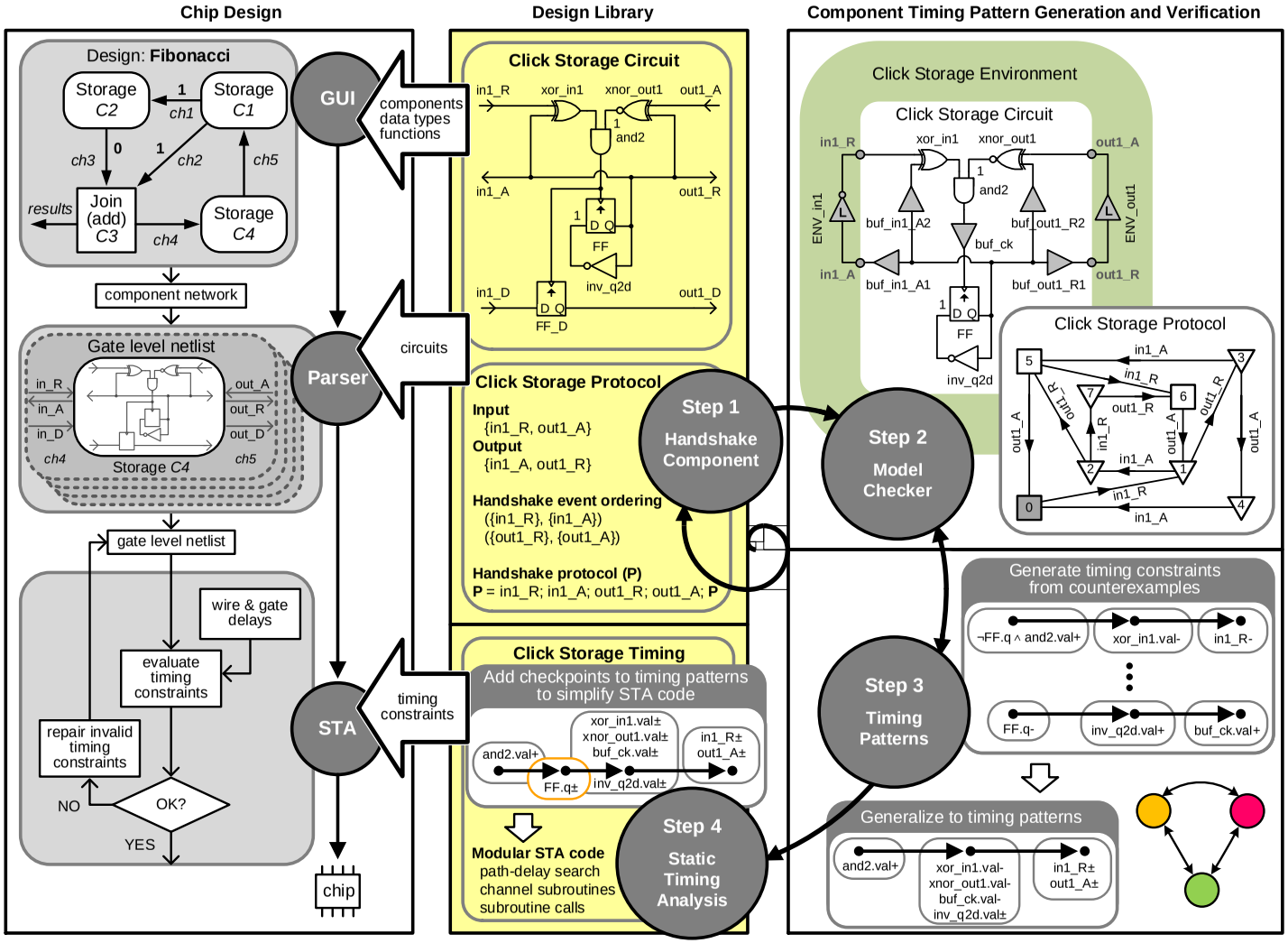


Figure 5.1: Reference diagram for this Chapter, illustrating the timing verification context and framework of ARCtimer. The *Design Library* in the center column connects the *Chip Design* flow on the left and the *Component Timing Pattern Generation and Verification* framework (ARCtimer) on the right.

protocols are delay insensitive, self-timed systems built using ARCtimer-verified components are also delay insensitive.

ARCtimer comes early in the design process as part of building a library of verified components for later system use. The library also stores *static timing analysis* (STA) code to validate and enforce the component's constraints in any self-timed system built using the library. The library descriptions of a handshake component's circuit, protocol, timing constraints, and STA code are robust to circuit modifications applied later in the design process by technology mapping or layout tools. This Chapter identifies critical choices and explains what modular timing verification entails and how it works.

Verifying gate-level signals against a handshake protocol is to identify and verify the essential internal timing constraints that make or break the component's protocol description.

We introduce ARCtimer, a framework set up precisely to identify internal timing constraints. ARCtimer targets pattern-based circuit families of handshake components – circuit families that use design patterns to describe the circuit implementations of their components. Families that do so include Micropipeline [54], Tangram and Balsa and Handshake Solutions [3, 11, 46], GasP [53, 55], QDI with precharge buffers [3, 23, 43], Mousetrap [45], and Click [36].

5.1 Timing Verification Context

Figure 5.2 shows three stages in a typical chip design flow for self-timed circuits. The stages are marked with the keywords *GUI* (Graphical User Interface), *Parser*, and *STA* (Static Timing Analysis). Other stages, for instance simulation and testing and layout placement and routing, are omitted. Each stage receives

information from the *Design Library*.

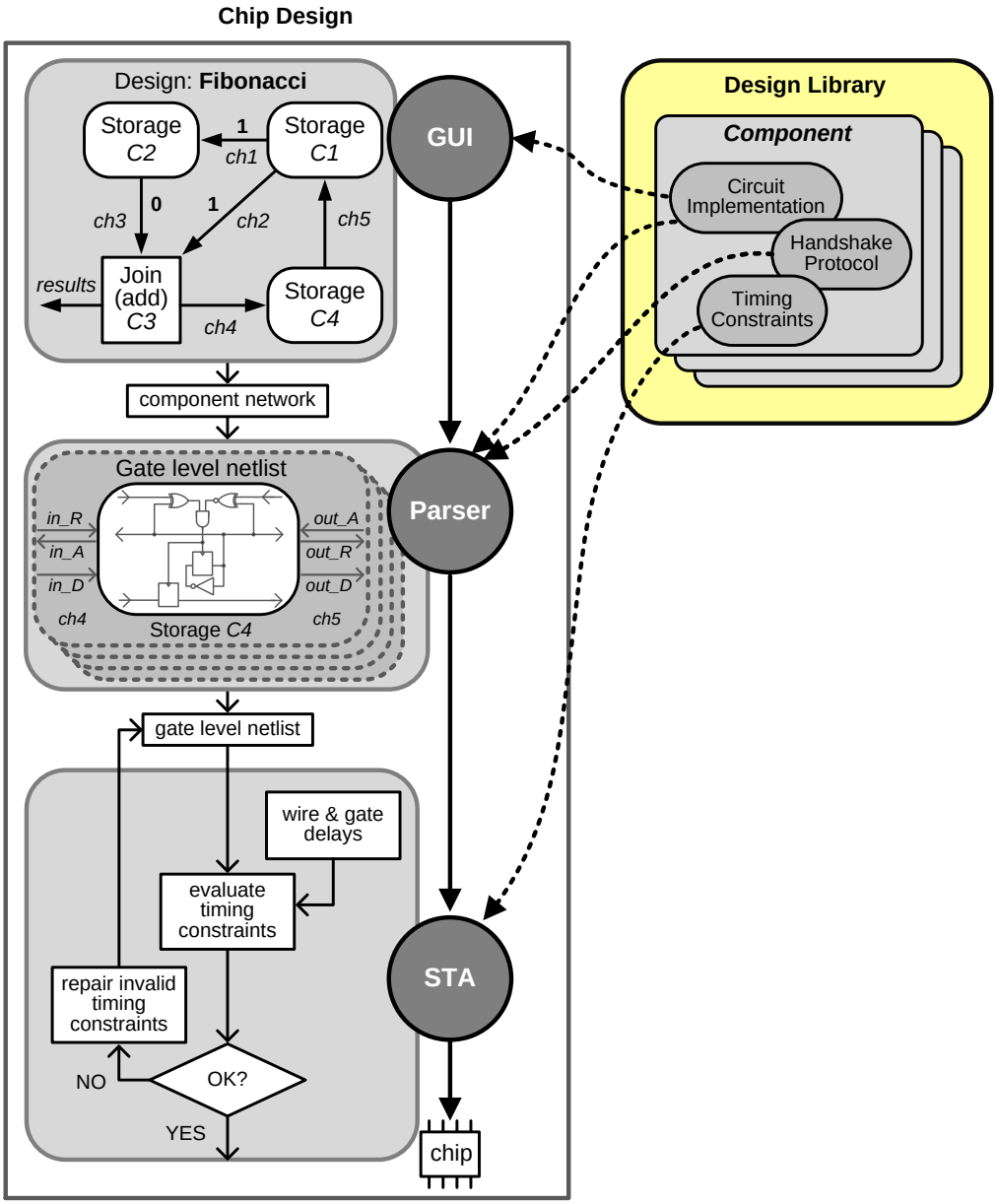


Figure 5.2: Overview of a typical chip design flow for self-timed circuits, showing an example of a Fibonacci number generator design. Simulation and testing and layout placement and routing are omitted. Each part of this Figure is discussed in more detail in the following subsections.

5.1.1 Design Library

An ideal design flow would support a variety of circuit families that could be mixed and matched based on the desired speed, power, energy efficiency, time-to-market or backward compatibility needs for the system or sub-systems. The Design Library for such a flow should store GUI, circuit, and protocol descriptions for the components of each family. Such a library should also store the timing constraints for each component.

5.1.2 GUI

Using a GUI (Graphical User Interface) or an equivalent written user interface, one can formulate a network of components connected by handshake channels. The GUI design in Figure 5.3 connects four components assembled to generate the Fibonacci sequence 1, 2, 3, 5, 8, etc on the *results* channel.

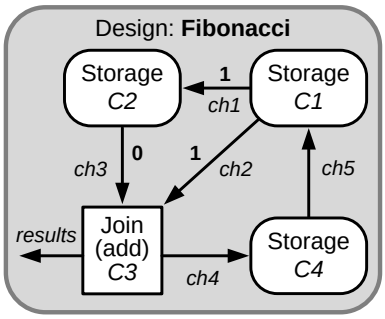


Figure 5.3: Fibonacci design using a GUI. Components are connected by handshake channels. Arrows indicate the direction of data.

Our GUI-formulated designs use function calls to represent data operations and a handshake protocol based on full and empty channels with data types. A full channel has valid data; an empty channel has data not yet valid or no longer used. The Storage components in the Fibonacci design act when their incoming channels are full and their outgoing channels are empty. When they act, they:

- *copy* the incoming data and forward the copied data,
- *fill* their outgoing channels, making them full, and
- *drain* their incoming channels, making them empty.

The Join component in Figure 5.3 adds the numeric data on its two incoming channels and forwards the sum. Having no storage facility for data, it waits to drain its incoming channels until all its outgoing channels are empty. This ensures that the incoming data remain stable until the sum is stored and acknowledged.

The Fibonacci design starts with all channels empty except for channels *ch1*, *ch2* and *ch3* that start full with initial data values respectively *1*, *1*, and *0* – as indicated in Figure 5.3 and Figure 5.4 (a). The Join forwards the sum of *0* and *1*, i.e. *1*, both to the *results* channel and to channel *ch4* going into Storage component *C4*, shown in Figure 5.4 (b). Storage *C4* forwards the Fibonacci result to Storage *C1*, and in doing so it fills *ch5* and drains *ch4*, shown in Figure 5.4 (c). This enables the Join to drain channels *ch2* and *ch3*, thus enabling Storage *C2* to act, shown in (d)–(e). *C2* acts by storing the data value *1* proffered over *ch1* and sending it on to *ch3*, thereby making *ch3* full and *ch1* empty, shown in (f). This in turn enables Storage *C1* to store and forward the new Fibonacci result *1* onto *ch1* and *ch2*, fill *ch1* and *ch2*, and drain *ch5*. The design is now back in a state similar to its initial state, with all channels empty except for *ch1*, *ch2*, and *ch3* that have the next set of data values: *1*, *1*, *1* respectively. The Join's next Fibonacci result will be *2*.

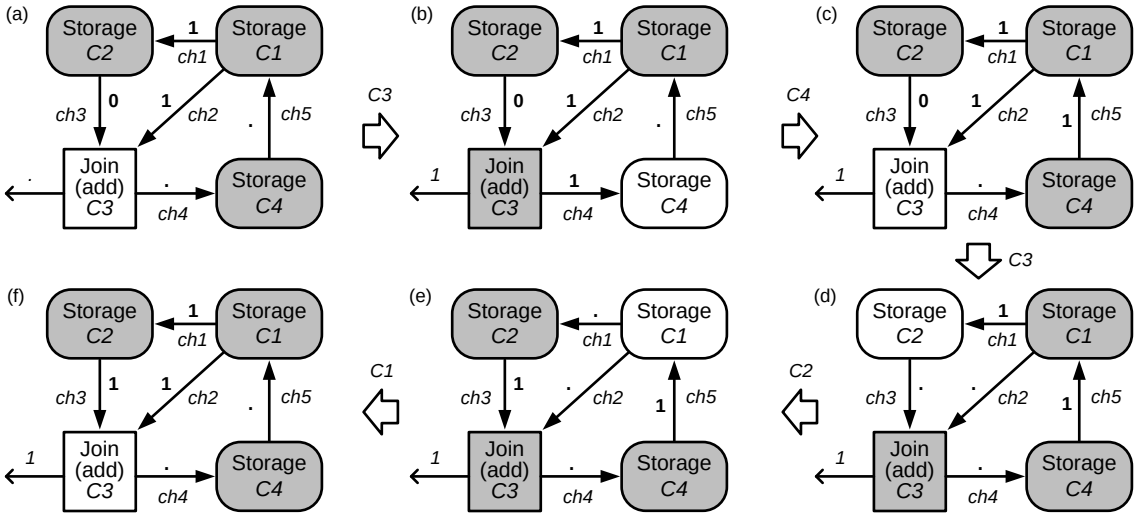


Figure 5.4: Fibonacci circuit example. Empty (drained) channels are marked with "." over the channel. For this Figure only, components are gray colored if no handshake events are possible. Steps (a) through (f) shows the *results* changing from no data to 1. Step (f) is similar to (a), but with new data for Join-*C3*

5.1.3 Parser

The Parser takes as input a component network from the GUI and expands it into a gate-level netlist for the protocol and circuit family selected by the user.

From the Fibonacci design in Figure 5.3, we chose a bundled-data two-phase non-return-to-zero (non-RTZ) handshake protocol, which uses a request wire, an acknowledge wire, and a bundle of wires with data. The gate-level netlist for Storage *C4*, shown in Figure 5.5, belongs to the Click circuit family [36].

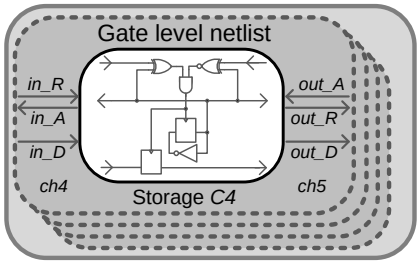


Figure 5.5: The GUI formulated Fibonacci design is parsed into a gate-level netlist.

There are several choices for expanding data functions, like the *add* function in the Join. One choice is to keep them as function calls. Standard hardware description languages, such as Verilog, can mix structural and functional descriptions [41]. Another choice is to expand the datapath circuits separately and organize the GUI formulation to optimize the flow of data. Standard design compilers excel at automatically synthesizing combinational functions into gate-level netlists. Automatically synthesizing sequential functions is more difficult, but possible when the goal is to optimize worst-case performance. However, a major promise of self-timed design is the ability to optimize average-case performance — in terms of latency, throughput, power, energy, or any combination thereof. Partitioning sequential functions into combinational functions that optimize *average-case rather than worst-case* performance has thus far eluded design automation. Such partitioning remains a collaborative effort between the designer and his or her design compiler [5, 17, 42, 44, 51].

5.1.4 STA

Static timing analysis (STA) [40] allows one to validate and repair timing constraints in the gate-level netlist generated by the Parser. Well-known examples of timing constraints for latches and flipflops are minimum clock pulse width, setup time, and hold time. A self-timed Design Library also holds relative timing constraints between end signals on paths that start at the same point but must arrive at their end points in a pre-established sequence. The delay slack in each constraint is parametrized and filled in during technology mapping.

A *Technology Library* for the chosen fabrication process will fill in further details on gate and wire delays, minimum clock pulse widths, etc. By using

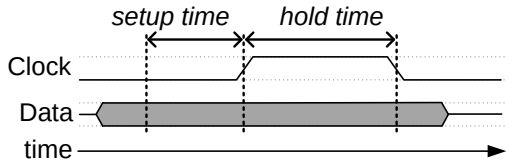


Figure 5.6: In Click circuits, data are captured at the rising clock edge as depicted in this timing diagram. Setup time is the amount of time from when the data must be ready before the rising clock edge. Hold time is the amount of time that the data must remain stable after the rising clock edge. Another way of describing the hold time is to say that the active clock edge has to arrive before releasing the data.

timing information stored in the Technology Library with physical information obtained from the chip, STA tools can compute and compare actual clock pulse widths against required minimum clock pulse widths, and add extra delay to repair inadequate pulse widths. The repairs go into the next chip layout iteration. STA tools can also repair relative timing constraints by adding sufficient delay to the "late path" with the pre-established later arrival.

There are several STA decisions that one must make, each with its own choices. Below, we will emphasize three important STA decisions, and indicate the choices that we have made.

The first STA decision to make is *where to insert delay* to repair an invalid timing constraint. One could insert the delay at the end point of the pre-established later end signal. Alternatively, one could insert the delay at a design-friendly location that might be exercised less frequently per protocol cycle and therefore retard the circuit performance less. Or one could choose a repair point that is shared by multiple invalid constraints, thus reducing the need to insert multiple delays.

We have chosen to specify a design-friendly delay insertion point for each timing constraint. Each constraint stored in our Design Library identifies a delay insertion point to use for its repair. The Design Library may indicate that the delay is

symmetric or that it retards only rising or only falling signals.

We formulate timing constraints from the viewpoint of a handshake component, even though the constrained paths may start or briefly wander outside the component. The STA code for a timing constraint stored in the Design Library records when and where a constrained path enters and exits the component. The "when" relates to a pre-established path signal sequence. The "where" is always a handshake signal because all components connect only through handshake channels. The STA code can identify a constraint with an external start point by identifying the two handshake signals that jointly started there.[1] Armed with this information, an STA tool can instantiate the STA code stored in the Design Library, fill in the sub-paths that are outside the component instance in the gate-level netlist, and complete the path-finding process in a modular fashion.

The second and equally important STA decision to make is *when to insert delay*. The many timing constraint instances associated with a gate-level netlist might not be independent to each other. Inserting delay to repair one invalid constraint instance may repair or invalidate others.

We use an iterative process similar to [36] for delay insertion. During STA, we group timing constraint instances that share the same delay insertion point instance[2] for repair. For each delay insertion point and its group of constraints, we maintain:

[1] We use this, for instance, to formulate bundled-data setup time constraints. Data flipflop *FF_D* in the Storage component in Figure 5.7 (left-column-top) has a setup time constraint with an external start point identified as the point where handshake signals *in1_R* and *in1_D* jointly started.

[2] We may use "constraint" and "insertion point" when it is clear from the context that we mean "constraint instance" and "insertion point instance."

- a list with delays of the constraints in the group,
- the maximum delay in the list, and
- the sum of the delays in the list.

The delay value of a constraint indicates the least delay one must insert into the gate-level netlist to make the constraint valid. The STA process stages delay insertion iteratively, inserting more delay at only one insertion point per iteration. As mentioned earlier, constraints are not necessarily independent, and so inserting more delay into the netlist to repair one constraint may repair others as well, or possibly damage them. Therefore, after each iteration the STA process re-computes the delay requirements for all constraints. The process is as follows:

- Start the first STA iteration, with all delays set to zero.
- After each iteration, update the information for each insertion point. For valid constraints, set the delay to zero. For invalid constraints, set the delay to a re-computed minimum delay mismatch rounded up to the best suitable delay device available in the Technology Library.
- If all groups have a maximum delay of zero then all timing constraints are satisfied, iteration ends, and the netlist can proceed to the next stage in the chip design.
- If one or more groups have non-zero delay, another iteration begins by adding delay to the worst offender. As worst offender, our process chooses an insertion point from those with the highest delay sum. The added delay is the maximum delay listed for this worst offender.

This iterative process may temporarily decrease the number of valid constraints from iteration to iteration, but it will converge unless constraints are circularly dependent, which rarely happens. Circularly dependent constraints force one to choose different delay insertion points or even different timing constraints.

Having discussed *where to insert delay* and *when to insert it*, we now come to the third and most important STA decision to make: *what STA engine to use*. Conventional STA tools are difficult to use on self-timed circuits because such tools fail to handle logic loops gracefully. Simple treatment of such loops is acceptable for the conventional design process because they are rare in clocked systems — loops in clocked systems tend to start and end at flipflops. Self-timed circuits, however, are rich with logic loops, as they must be because the unstable behavior of closed logic loops animates self-timed behavior.

Graceful analysis of rise and fall times and delay of gates in logic loops requires a two-pass process. A first pass computes output rise and fall times from gate size, gate load, and input rise and fall times. This pass converges very quickly because output rise and fall times are a very weak function, almost independent, of input rise and fall times. A second pass computes the delay of each gate using the input rise and fall times from the first pass.

Conventional STA tools combine those two passes into one concurrent process. They split loops into linear acyclic paths to make a one-pass estimation effective. Moreover, they commonly use a "clock," rare in self-timed circuit designs, to guide where to split each loop. Some self-timed design groups have invested heroic effort in fresh ways to split loops in order to apply conventional STA tools to self-timed systems [2, 36, 38, 52, 61], but none work truly gracefully.[3]

[3]This applies also to the Click self-timed circuit family, which was developed specifically to work with conventional STA and test tools [36]. Click circuits use only flipflops as state-holding

The time has come to use a two-pass process to analyze loops intact. Loops are, after all, central to self-timed circuit design.

Our STA engine is set up to work self-standing or with an existing STA tool. Its internal algorithms to find paths and calculate path delays are still too coarse-grained to replace existing STA tools, but adequate for early design exploration. We use the STA engine in self-standing mode to evaluate the timing in new handshake components before we have formalized timing constraints using ARCtimer — the timing verification framework discussed in the next section, Section 5.2. We use the self-standing mode again to validate the STA code for the timing constraints produced by ARCtimer and stored in the Design Library. By inserting pseudo-random delays at multiple pseudo-randomly selected points in the netlist we force the STA engine to recompute compensating delays, and then we simulate and test the repaired netlist for correct functionality.

5.2 Timing Verification Framework

The spiral in Figure 5.7 shows the four main steps in our timing verification framework[4] for handshake components. We call this framework *ARCtimer*. The steps use the keywords: *Handshake Component* (Step 1), *Model Checker* (Step 2), *Timing Patterns* (Step 3) and *Static Timing Analysis* (Step 4). Step 1 begins and Step 4 ends in the left column with the Design Library of component descriptions for each circuit family supported by the design flow. This Figure illustrates the steps for a Click Storage component with single incoming and outgoing channels.

elements, and have a flipflop in every loop. Some Click loops, however, go *through* flipflops and fail to start or end at flipflops. Conventional STA tools require splitting such loops.

[4]We use the term "framework" because we already reserved the term "system" for large-scale designs, and because the term "flow" is often associated with automatic solutions and we seek to avoid that connotation.

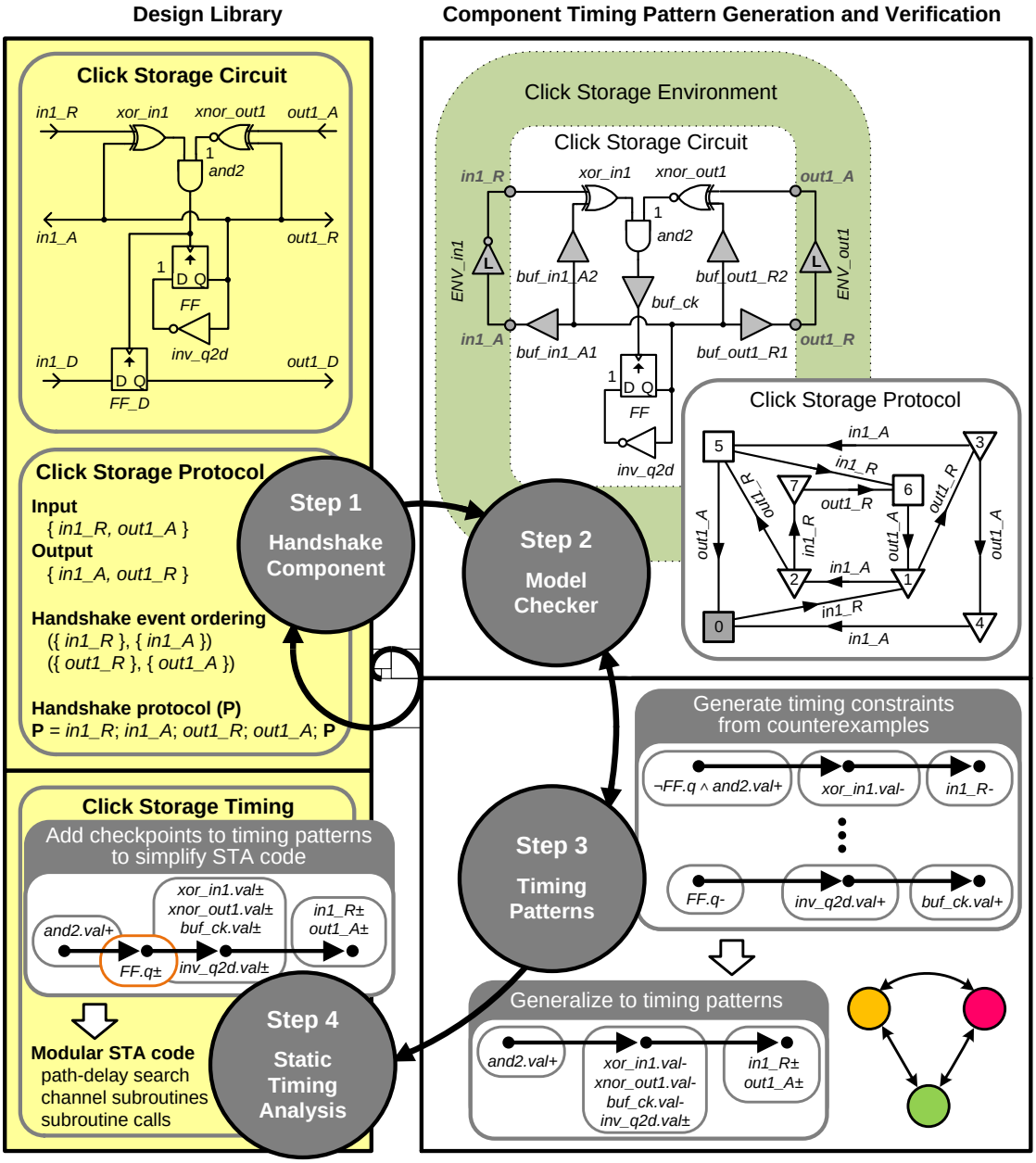


Figure 5.7: The ARCtimer framework has four main steps. Starting from a circuit implementation and XDI description of the specification, it goes through a model checker – finding timing constraints, and generalizing them for use with STA tools. The whole process starts and ends in the Design Library.

We use this framework in two ways, with and without priming. Without priming, ARCtimer takes the circuit and protocol descriptions of a component

and helps us uncover all the timing constraints. The set of timing constraints thus produced ensures that the circuit obeys the protocol. ARCtimer works well without priming for simple components such as the Storage and Join in the Fibonacci design in Figure 5.3. For complex, nondeterministic, or data-driven components the run time and space limitations of underlying tools may necessitate priming ARCtimer with a starter set of timing constraints and using ARCtimer to complete the set.

The sub-sections below explain each step in more detail.

5.2.1 ARCtimer Step 1 — Handshake Component

A handshake component responds to the full and empty state of its channels, as we illustrated earlier in Section 5.1.2 for the Storage and Join components in the Fibonacci design.

The circuit-level representations for full and empty channels depend on the variant of the handshake protocol used. Many circuit families, including Click [36], Micropipeline [54], and Mousetrap [45] use a two-phase non-return-to-zero (non-RTZ) protocol with separate request and acknowledge wires to encode full or empty. GasP uses a two-phase return-to-zero (RTZ) protocol [53,55] with a single statewire to represent full or empty. Figure 5.8 shows the default representations for full and empty in two-phase non-RTZ and two-phase RTZ handshake protocols.

In general, the control logic of a handshake component is an AND function of the conditions necessary for it to act. Complex handshake components may have multiple such AND functions to guard different actions. The Click Storage component in Figure 5.9 has one such AND function — labeled *and2*.

The response of a handshake component usually changes the state of one or more of the channels to which it responded. Many components drain full incoming

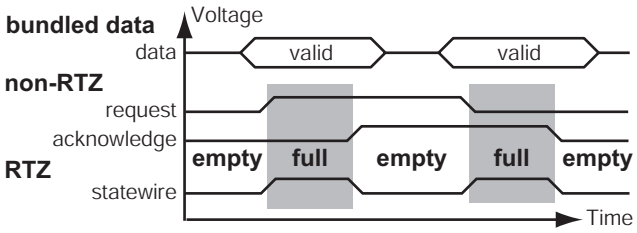


Figure 5.8: Default state representations for full and empty channels in two-phase non-RTZ and RTZ handshake protocols with bundled data. A channel with non-RTZ protocol is engaged in a handshake, i.e. is full, when its *request* and *acknowledge* differ. A channel with RTZ protocol is full when its *statewire* is high. During the handshake, i.e. when the channel is full, data must be valid and remain stable.

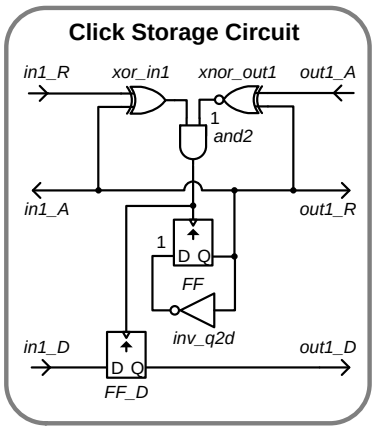


Figure 5.9: Circuit implementation of Click Storage including the data path. Initially, all wires of the control path have a logical value of 0, except for the wires marked 1.

channels and fill empty outgoing channels. Thus, there is a feedback loop from channel state to component action to channel state. The Click Storage component in Figure 5.9 has two such loops: one for channel *in1* from $in1\_R$ through gates xor_in1, *and2*, *FF* to $in1\_A$; another for channel *out1* from $out1\_A$ through gates $xnor\_out1$, *and2*, *FF* to $out1\_R$.

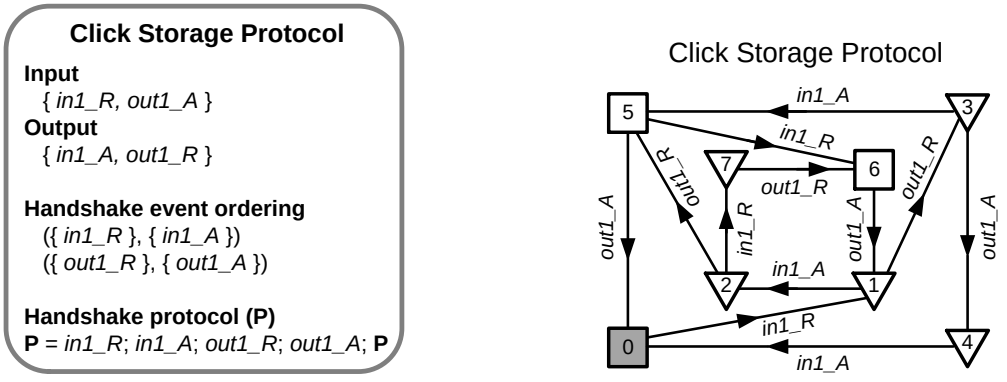
The AND function coordinates the two loops and makes the Click Storage component "act." The component's gate-level actions are similar but more refined than its GUI-level actions described in Section 5.1.2. The action triggers when *in1* is full ($in1\_R \neq in1\_A$) and *out1* is empty ($out1\_R = out1\_A$) — see Figure 5.8.

When detected, these cause rising transitions on *xor_in1* and *xnor_out1* that in turn cause AND function *and2* to rise. A rising transition on *and2* clocks the edge-triggered flipflops and starts three actions concurrently:

- *FF_D* captures and copies data from *in1_D* to *out1_D*.
- *FF* inverts the value on signal *in1_A*, thus draining *in1*.
- *FF* also inverts the value on *out1_R*, thus filling *out1*.

The now empty *in1* and now full *out1* reset *xor_in1* and *xnor_out1* to low, each of which resets *and2* to low, thus bringing the Click Storage circuit back to an initial state where it can coordinate the next full *in1* and empty *out1* handshakes.

We initialized the Click Storage circuit which is part of the Design Library as shown in Figure 5.9 with all channels empty. All its signals have a logical value of 0, except for the output of *xnor_out1* and the *D* input of *FF* which are 1, as indicated. This initial state in the *Storage circuit* matches the initial state of the grey-colored state 0 in the corresponding finite state machine protocol specification in Figure 5.10b.



(a) Handshake protocol in XDI **(b)** Translated in to FSM

Figure 5.10: Click Storage Protocol and translation in to FSM.

One can choose various specification formalisms to describe the protocol behavior of a single handshake component or of a self-timed network of handshake components. Dialects of *Communicating Sequential Processes* (CSP), sometimes called *Communicating Hardware Processes* (CHP), are very popular [3, 46]. The *Calculus of Communicating Systems* (CCS) forms the basis of the self-timed circuit verification work in [52,62]. *Signal Transition Graphs* and *Petri Nets* form the basis of the self-timed circuit verification and synthesis work in [8, 19, 25].

The goal of this Chapter is merely to show how to verify single components. We consider here neither how to synthesize a component nor how to verify networks of them. This limited goal gives us the leisure of selecting a formalism whose specifications are both *compact*, i.e. short and easy to understand, and *complete*, i.e. fully delay-insensitive. We found a suitable formalism in the theory of *Delay-Insensitive Algebra* developed by [14, 20, 59]. Delay-Insensitive Algebra also underlies [30] which uses it to build a verification framework for self-timed circuits. Our goal is much simpler than any of the synthesis and verification work built on Delay-Insensitive Algebra. We merely seek compact and complete specifications that allow us to verify that a component's circuit has the same choices of action as specified by the component's protocol.

Delay-Insensitive Algebra uses finite traces of events that specify not only safety properties, but also liveness properties that are crucial for distinguishing choices of action. It uses an interleaving semantics that represents parallel events by ordering them arbitrarily.

The protocol description in Figure 5.10a first identifies the signals coming into the Click Storage (Input) and those going out (Output). This information will be used to complete the compact description into a fully delay-insensitive one. Next

come the handshake event orderings for the two channels. Each event is either a rising or a falling signal transition. Each channel of the Click Storage component starts with an event on its request signal, and thereafter alternates events on its request and acknowledge signals, This corresponds to the basic two-phase non-RTZ handshake communication protocol for an initially empty channel, illustrated in Figure 5.8. Last comes the protocol description P — a compact repetitive sequence of four consecutive input-output events:

$$P = in1\_R \ ; \ in1\_A \ ; \ out1\_R \ ; \ out1\_A \ ; \ P.$$

In this form, protocol P says that the Click Storage component must wait for input event $in1\_R$ before it produces output event $in1\_A$ followed by output event $out1\_R$, after which it waits again until it receives another input event, namely $out1\_A$, before it repeats the same protocol, P.

The delay-insensitive interpretation of P allows more behaviors. The interpretation is based on what is popularly known as the *Foam Rubber Wrapper* metaphor, a term for delay-insensitive communication introduced by the late Charles Molnar. The idea is that an event may be delayed for an arbitrary time when it travels between sender and receiver components. Thus, an input event in an event sequence specified by P might have occurred as early as its generation or as late as its receipt, or anywhere in between. Hence, input events $in1\_R$ and $out1\_A$ in P may move to earlier positions in the sequence provided each input follows the previous output event on the same channel, as specified in the handshake event orderings. Likewise, output events $in1\_A$ and $out1\_R$ may move to later positions in the sequence provided each output precedes the next input event on the same channel.

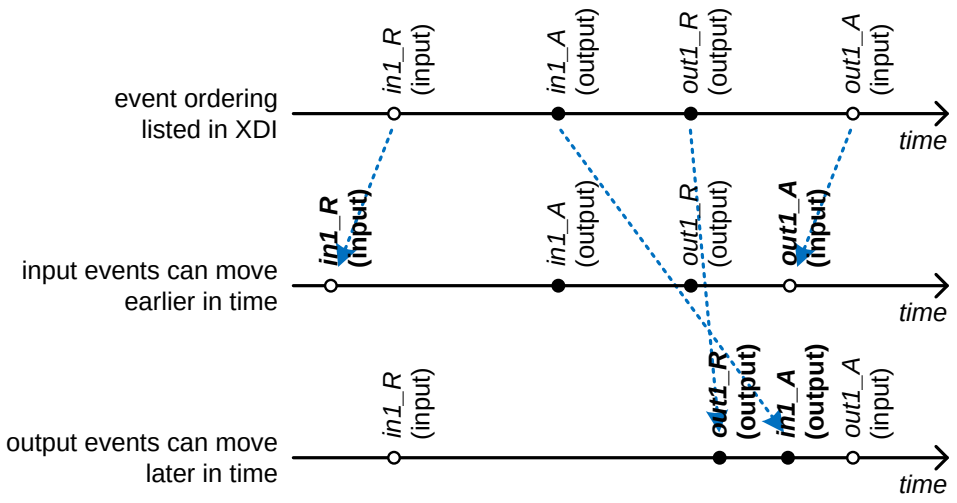


Figure 5.11: Possible event orderings. For better visibility, input events are colored with white dots and output events are colored black dots. The events which moved position in time are highlighted with bold font. From the top of the Figure where there is a sequence of four events, input events can move backward in time (middle) and output events can move forward in time (bottom), provided they follow the handshake event ordering specified in Figure 5.10a.

We use tools developed for Delay-Insensitive Algebra in [20] to complete the compact protocol description expressed as P automatically into a fully delay-insensitive description expressed as the finite state machine in Figure 5.10b.

The finite state machine in Figure 5.10b describes the various event sequences and event choices at the pair of channel interfaces of the Storage component. It also describes the progress expectations at each state in an event sequence. The triangles ($\bigtriangledown$) denote transient states that may persist only for a finite time. Triangular states typically respond to handshake output events, which are controlled by the component. The underlying assumption is that the internal circuit actions leading up to the output event will finish within a finite amount of time. This is valid for most actions, with the possible exception of non-deterministic arbitration — absent from a Storage component. The rectangles ($\square$)

denote non-transient states that may persist forever. Rectangular states typically produce only input events — events controlled by the component's environment. The underlying assumption is that the environment might be lazy and never act. The finite state machine constrains the component to exit a transient state within unbounded but finite time, but allows it to remain in a non-transient state forever.

Note that these descriptions can be used for any Storage component with single incoming and outgoing channels and two-phase non-RTZ handshakes. One can easily envision how to generalize both descriptions to arbitrary numbers of channels. Other handshake components, such as the Join in the Fibonacci design of Figure 5.3, and even non-deterministic and data-driven components, also have relatively simple compact descriptions that are easy to understand [20].

The combination of a compact protocol description, P, and tool automation to complete P into a fully delay-insensitive description helps avoid over-specifying components. Avoiding over-specification is important and harder than one might think. We inadvertently and repeatedly over-specified the handshake behavior of a component using the approach in [52, 62], which requires complete specifications in CCS without tool support to help make them.

5.2.2 ARCtimer Step 2 — Model Checker

Figure 5.10 illustrates how one can model the protocol of a handshake component as a finite state machine. The machine serializes sequential as well as parallel events and captures the serialized behavior in event-based state transitions, state transition choices, and transient and non-transient states. Similar finite state machine descriptions can model gates, wires, the networks of gates and wires that form the circuit of a handshake component, and even the timing constraints

of a handshake component. Verifying that the component's circuit meets the component's protocol under the component's given set of timing constraints thus becomes a model checking task [7].

We experimented with a customized model checker, *Analyze* [52], as well as a general-purpose model checker, *NuSMV* [6]. We have found customized model checkers especially hard to use for modeling and verifying the protocols and circuits of non-deterministic and data-driven handshake components. Moreover, we mistrusted some of the verification results that we obtained. We resolved the difficulty in modeling the protocols by using formalisms and tools developed for Delay-Insensitive Algebra, as explained in Section 5.2.1. Other difficulties vanished with use of a general-purpose model checker. General-purpose model checkers force one to indicate explicitly both what to verify and how to execute the various parts of a model. Although explicitness requires more work, it gives one full control over one's own experiments.

The experiments and code fragments shown in this Chapter are based on NuSMV [6], a model checker that is freely available and has an active and diverse user community. NuSMV has helped us generate and verify timing constraints for widely different components with deterministic, non-deterministic, and data-driven handshake behaviors. The timing verification work in [9] use NuSMV but verify fewer properties than we do, as we will explain in Sections 5.2.2.1–5.2.2.2.

Figure 5.12 shows what a general-purpose model checker must have and do to verify a handshake component's circuit against its protocol under a given set of timing constraints. Note that besides models for the circuit, protocol, and timing constraints, there is the component's environment — a model for the environment in which the component's circuit operates. We model the component's environment

by providing a separate interface for each channel that responds to channel outputs in any of all the valid ways possible for that channel.

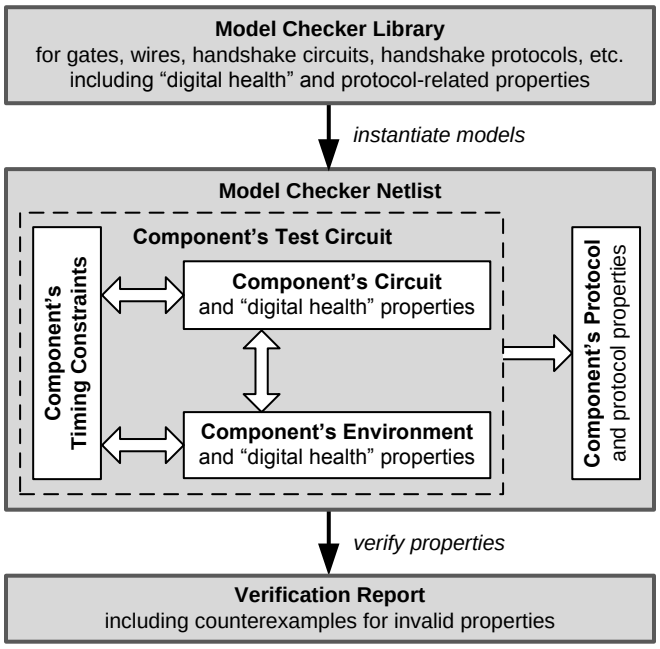


Figure 5.12: Organization of the model checking task to verify, for a given handshake component, that the component's circuit in its environment and under its timing constraints satisfies both the gate-level "digital health" properties and the properties defined by the component's protocol. Examples of "digital health" are semimodularity, explained earlier in Section 3.7, and absence of set-reset drive fights.

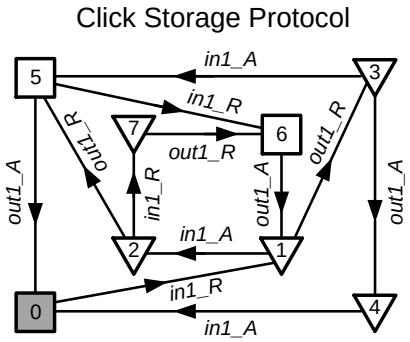
From Figure 5.12, the grey rectangle at the top represents the *Model Checker Library* — a translation of the Design Library in Figure 5.7 (left-column) into model checker lingo. Using the Model Checker Library, ARCtimer creates a *Model Checker Netlist* represented by the middle grey rectangle. The netlist connects single instances of the component's protocol, circuit, environment, and available timing constraints, as indicated by the white and broken-line rectangles and the white arrows. A white arrow follows the direction from a rectangle with models that create an event to a rectangle with models that respond to that event. All events in and between *rectangles with horizontal text* are interleaved using an

asynchronous mode of operation. Events of *rectangles with vertical text* must be synchronized to corresponding events, which is achieved by operating them in synchronous mode. The model checker takes the Model Checker Netlist and first generates a corresponding finite state machine model with instantiated gate-level "digital health" and protocol-related properties for verification, and then it checks the properties. The grey rectangle at the bottom represents the verification report with a pass or fail indication per property and a counterexample of a computation path in the resulting finite state machine for each failing property.

The following sub-sections give a more detailed explanation of Figure 5.12, including code fragments with NuSMV solutions.

5.2.2.1 Modeling the Component's Protocol

Figure 5.13a repeats the complete, fully delay-insensitive protocol specification of Figure 5.10b and shows its translation into NuSMV model checker lingo.



(a) Fully delay insensitive protocol specification

```
1  MODULE protocol (in1_R, in1_A, out1_R, out1_A)
2    VAR
3      state: {s0, s1, s2, s3, s4, s5, s6, s7, errorOUT, errorIN};
4    ASSIGN
5      init(state) := s0;
6    TRANS
7      next(state) = case
8        --legal handshake transitions
9        state = s0  & (in1_R  != next(in1_R)) : s1;
10       state = s1  & (in1_A  != next(in1_A)) : s2;
11       state = s1  & (out1_R != next(out1_R)): s3;
12       state = s2  & (in1_R  != next(in1_R)) : s7;
13       state = s2  & (out1_R != next(out1_R)): s5;
14       state = s3  & (out1_A != next(out1_A)): s4;
15       state = s3  & (in1_A  != next(in1_A)) : s5;
16       state = s4  & (in1_A  != next(in1_A)) : s0;
17       state = s5  & (in1_R  != next(in1_R)) : s6;
18       state = s5  & (out1_A != next(out1_A)): s0;
19       state = s6  & (out1_A != next(out1_A)): s1;
20       state = s7  & (out1_R != next(out1_R)): s6;
21       --illegal handshake transitions
22       in1_A  != next(in1_A) | out1_R != next(out1_R) : errorOUT;
23       in1_R  != next(in1_R) | out1_A != next(out1_A) : errorIN;
24       --remaining transitions
25       TRUE: state;
26     esac;
27
28   --PROPERTIES
29    --safety
30    CTLSPEC AG state != errorOUT
31    CTLSPEC AG state != errorIN
32    --progress
33    CTLSPEC AG (AF (state!=s1))
34    CTLSPEC AG (AF (state!=s2))
35    CTLSPEC AG (AF (state!=s3))
36    CTLSPEC AG (AF (state!=s4))
37    CTLSPEC AG (AF (state!=s7))
38    --choice equivalence
39    CTLSPEC AG (state = s0 -> E[state = s0 U state = s1])
40    CTLSPEC AG (state = s1 -> E[state = s1 U state = s2])
41    CTLSPEC AG (state = s1 -> E[state = s1 U state = s3])
42    CTLSPEC AG (state = s2 -> E[state = s2 U state = s7])
43    CTLSPEC AG (state = s2 -> E[state = s2 U state = s5])
44    CTLSPEC AG (state = s3 -> E[state = s3 U state = s4])
45    CTLSPEC AG (state = s3 -> E[state = s3 U state = s5])
46    CTLSPEC AG (state = s4 -> E[state = s4 U state = s0])
47    CTLSPEC AG (state = s5 -> E[state = s5 U state = s6])
48    CTLSPEC AG (state = s5 -> E[state = s5 U state = s0])
49    CTLSPEC AG (state = s6 -> E[state = s6 U state = s1])
50    CTLSPEC AG (state = s7 -> E[state = s7 U state = s6])
```

(b) Corresponding NuSMV code

Figure 5.13: The first part of the code in lines 1–26 describes legal and illegal states and transitions. The model checker uses this part to monitor the sub-system with circuit, environment, and timing constraints. The second part in lines 28–50 describes the protocol properties for "what the monitor must see." **Note:** A NuSMV case statement gives higher priority to the guarded commands in earlier lines of the case statement.

The translation is wrapped in a self-contained module, with the abbreviated name *protocol*, with formal parameter names for the handshake signals. The module's full name is *Click_Storage_1_In_1_Out_Protocol*.[5] We store such modules in the Model Checker Library — top box in Figure 5.12.

The first part of the translation, up to line 26 in Figure 5.13, codes the states, initial state, and event-based state transitions of the protocol. Each translated state name begins with the letter *s* followed by the original state number — e.g. initial state *0* in Figure 5.13a translates to *s0* in Figure 5.13b. The original protocol specifications in Figure 5.10 specify only legal states and transitions, omitting illegal and irrelevant ones. The omissions must be coded, however. We code two types of error states to receive illegal handshake transitions: illegal channel outputs go to *errorOUT*, and illegal channel inputs go to *errorIN*. All other events, irrelevant to the protocol, preserve the protocol's state. The resulting code forms a monitor. It will be used to monitor the *Component's Test Circuit* — the subsystem inside the broken line in Figure 5.12 (middle) which holds the component's circuit, environment, and timing constraints.

To monitor the Component's Test Circuit the protocol operates in synchronous mode. This means that the protocol's finite state machine code is executed in each execution step by the model checker. NuSMV uses the keyword *TRANS* in line 6 of Figure 5.13b to indicate that the next statement is to be executed in synchronous mode. The next statement, enclosed by the keywords *case* and *esac* in lines 7 and 26, is precisely the monitor code of the component's protocol in the rightmost white rectangle in Figure 5.12.

[5]Its un-abbreviated name says that the module has the protocol translation for a Click Storage component with 1 incoming and 1 outgoing channel.

The purpose of monitoring the Component's Test Circuit is to annotate its behavior for verification. Verification is done by checking properties. The properties in lines 28–50 of Figure 5.13b specify what the protocol must see when it monitors the Component's Test Circuit.

The properties in the second part of the code, lines 28–50, are inherent in the protocol specification, and translated along with the rest of the code. The two *safety* properties in lines 30–31 allow only legal handshake behaviors. The five *progress* properties in lines 33–37 allow the five transient states to persist for only a finite time. The transient states correspond to the triangles ($\bigtriangledown$) in the original specification. The remaining *choice equivalence* properties spell out the choices of action that must be available to the observed sub-system to meet the protocol specification. These might be refined with additional event information, if needed. The structure of these properties is quite straightforward for the Click Storage component, but becomes more interesting for non-deterministic components.

5.2.2.2 Modeling the Component's Circuit and Environment

Figure 5.14 repeats the gate-level Click Storage circuit and environment models in Figure 5.7 (right-column-top) and shows the corresponding gate-level NuSMV translation in Figure 5.15, using two gate models, *cgate* and *ff_posedge* that are defined in Figure 5.16.

The two translations, *circuit* and *environment*, are wrapped in self-contained modules with formal parameter names to support the exchange of handshake and timing constraint signals. These contain the code for the middle two white rectangles with horizontal text in Figure 5.12.

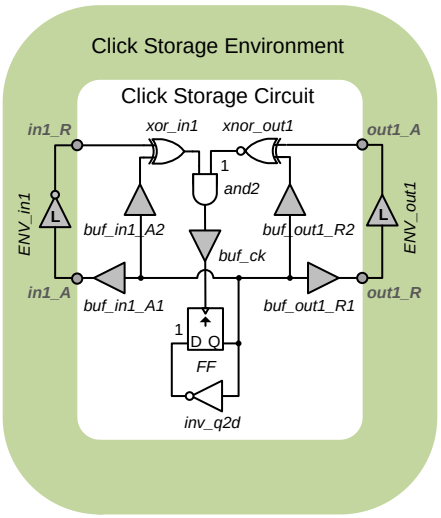


Figure 5.14: Click Storage circuit and environment models from Figure 5.7 (right-column-top). The gates for *ENV_in1* and *ENV_out1* contain the letter "L" to indicate that they are lazy.

The *Click Storage Circuit* shown in Figure 5.14 contains five more buffers than the original circuit description in the Design Library of Figure 5.7. The extra buffers are colored grey and named *buf_in1_A1*, *buf_in1_A2*, *buf_out1_R1*, *buf_out1_R2*, and *buf_ck*. The translation adds these buffers to delay wires and individual wire branches independently from gates. Buffers are necessary because the model checker ignores wire delays. Adding a buffer or inverter device to a logical wire connection makes that connection visible to the model checker as a device output with a device delay. It suffices to add buffers only to wires that branch out and to wires that clock edge-triggered flipflops. It is straightforward to adapt a compiler that generates the original circuit description to generate also the description for the model checker. The circuit description for the model checker also contains datapath signals *in1_D* and *out1_D* and datapath flipflop *FF_D*, which are omitted from Figure 5.14 to focus on the control logic and are postponed to Chapter 6.

```
1  MODULE circuit (in1_R, out1_A)
2    VAR
3      xor_in1     : process cgate (in1_R xor buf_in1_A2.val, f,f,f,f);
4      xnor_out1   : process cgate (out1_A xnor buf_out1_R2.val, t,f,f,f);
5      and2        : process cgate (xor_in1.val  & xnor_out1.val, f,f,f,f);
6      buf_ck      : process cgate (and2.val, f,f,f,f);
7      FF          : ff_posedge    (buf_ck.val, inv_q2d.val, f);
8      inv_q2d     : process cgate (!FF.q, t,f,f,f);
9      buf_in1_A1  : process cgate (FF.q, f,f,f,f);
10     buf_in1_A2  : process cgate (FF.q, f,f,f,f);
11     buf_out1_R1 : process cgate (FF.q, f,f,f,f);
12     buf_out1_R2 : process cgate (FF.q, f,f,f,f);
13   DEFINE
14     in1_A       := buf_in1_A1.val;
15     out1_R      := buf_out1_R1.val;
16     f           := FALSE;
17     t           := TRUE;
18   FAIRNESS running
19
20 MODULE environment (in1_A, out1_R)
21   VAR
22     ENV_in1    : process cgate (!in1_A, f,t,f,f);
23     ENV_out1   : process cgate (out1_R, f,t,f,f);
24   DEFINE
25     in1_R      := ENV_in1.val;
26     out1_A     := ENV_out1.val;
27     f          := FALSE;
28     t          := TRUE;
29   FAIRNESS running
```

Figure 5.15: Module definitions for the Model Checker Library. The code for *cgate* and *ff_posedge* follows in Figure 5.16.

Figure 5.15 shows the translated circuit module in lines 1–18 and the translated environment module in lines 20–29. Lines 3–12 describe the gate instances and their connections for the circuit. Lines 22–23 do the same for the environment. Most gates are instantiated as *process cgate (function,...)* where *function* is a Boolean logic combination of the module's parameters and outputs of other gates. The instances have the same names and logical functions as in Figure 5.14. For example, gate instance *xor_in1* in line 3 of Figure 5.15 computes the exclusive-or of parameter *in1_R* and *buf_in_A2.val*, the output of gate *buf_in_A2*. Likewise, positive edge-triggered flipflop instance *FF* in line 7 copies and stores the value on *inv_q2d.val* onto its output *q* whenever its clock input *buf_ck.val* changes from low (FALSE) to high (TRUE). The signal definitions in lines 14–17 and 25–28 following the keywords *DEFINE* serve to shorten and simplify various code fragments.

The operations of the circuit and its environment are monitored by the protocol

as explained in Section 5.2.2.1. Because we describe protocols with Delay-Insensitive Algebra, which uses an interleaving semantics, the protocol model interleaves its events. Thus, the protocol can interpret handshake events only when they arrive in sequence. Consequently, the circuit and its environment must interleave all handshake events because these are the events they share with the protocol. To simplify the overall execution, we chose to interleave not just the handshake events but all events generated by *cgate* instances in the circuit or its environment.[6] NuSMV pairs the keywords *process* and *cgate* in lines 3–12 and 23–24 in Figure 5.15 to indicate that the *cgate* instance is to be executed in asynchronous mode by interleaving its operations with those of other *process cgate* instances.

The asynchronous interleaving mode of operation comes with a cost of fairness conditions for selecting which *process cgate* operation to run next. The protocol assumes that most circuit operations take a finite time. It expects the circuit to generate a handshake output within a finite number of execution steps after receiving a handshake input from its environment. The NuSMV code for the protocol uses progress properties to formulate and verify these expectations — see lines 33–37 of Figure 5.13b. To satisfy these progress properties, each *process cgate* instance in the module must be selected to run after every so many unbounded but finite execution steps. The NuSMV statements *FAIRNESS running* in lines 18 and 29 of Figure 5.15 enforce precisely that.

The remaining code details can be explained by examining the module definitions for *cgate* and *ff_posedge* in Figure 5.16.

[6]This simple mode of interleaving can be combined with a *simultaneous* mode of operation [9] for internal gates that generate non-handshake events, allowing arbitrary subsets of these to operate simultaneously.

```
1  MODULE cgate (set, init_val, lazy, stop_rise, stop_fall)
2    VAR
3     val : boolean;
4     semimodular : boolean;
5    ASSIGN
6     init(val)          := init_val;
7     init(semimodular)  := TRUE
8     next(val) := case
9       (stop_rise & !val & set) | (stop_fall & val & !set)  : val;
10      lazy                                                 : {val, set};
11      TRUE                                                 : set;
12    esac;
13   TRANS
14     next(semimodular) = case
15     ((!stop_rise & !val & set) | (!stop_fall & val & !set))
16                 & next(val)=next(set) & next(val)=val = : FALSE;
17     TRUE                                               : semimodular;
18   esac;
19   --PROPERTIES for digital health
20   CTLSPEC AG semimodular
21
22 MODULE ff_posedge(ck, d, init_q)
23  VAR
24   q : boolean;
25  ASSIGN
26   init(q) := init_q;
27  TRANS
28   next(q) = case
29    !ck & next(ck) = : d;
30    TRUE  : q;
31   esac;
```

Figure 5.16: Module definitions for *cgate* and *ff_posedge*

The module definition of *cgate*, i.e. "combinational gate," follows in lines 1–20 of Figure 5.16. Each *cgate* takes an arbitrary Boolean combinational logic function through its first parameter, *set*. For example, the *cgate* for *xor_in1* in line 3 of Figure 5.15 takes *in1_R xor buf_in1_A2.val* — the exclusive-or of Boolean signals *in1_R* and *buf_in_A2.val*. The second parameter, *init_val*, contains the initial value of *cgate* output *val*, assigned in line 6 of Figure 5.16. For example, the output of *xnor_out1* in line 4 of Figure 5.15 is initialized to TRUE, which corresponds to the value *1* indicated for the *xnor_out1* output in Figure 5.14. When a *cgate* instance is selected to run, it evaluates its *set* function. Depending on the other input parameters in Figure 5.16, it either updates its output *val* with the *set* result (line 10 or 11) or does nothing (line 9 or 10). Only lazy or timing constrained *cgate* instances may do nothing.

A *cgate* is lazy if its third parameter, *lazy*, is TRUE. For example, both the Click Storage Environment gates *ENV_in1* and *ENV_out1* in lines 22–23 of Figure 5.15

are lazy. A lazy *cgate* has an arbitrary choice either to act by setting its output *val* to the result in *set* or to do nothing by keeping the old value of *val*. This nondeterministic choice is indicated in line 10 of Figure 5.16 by the curly brackets around *val* and *set*.

Timing constraints may prevent a *cgate* output transition from FALSE to TRUE (rise), from TRUE to FALSE (fall), or both. Output *val* cannot rise in line 9 of Figure 5.16 if the fourth parameter *stop_rise* is TRUE, and neither can it fall if the fifth parameter *stop_fall* is TRUE. In Section 5.2.3, we will discuss how timing constraints control the run-time values of *stop_rise* and *stop_fall* in the various *cgate* instances.

It is possible that a *cgate* instance, poised to have its output rise or fall, fails to be selected and do the output transition before a new *set* value arrives that disables the transition. For *cgate* instances used in self-timed circuits, the presence of a later *set* value overtaking an earlier one often indicates the presence of a race condition. We therefore flag such overtakings for later inspection. A variable with the name *semimodular*, initially TRUE (line 7), becomes FALSE at the first such overtaking (lines 15–16) when the next execution step no longer shows an enabled transition (*next(val)=next(set)*) but also shows no sign of having taken it (*next(val)=val*). The NuSMV model checker updates variable *semimodular* (lines 14–18) at each execution step, as indicated by the keyword *TRANS* in line 13. The "digital health" property in line 20 requires *semimodular*[7] to be TRUE at all times, and flags any change to FALSE.

Variable *semimodular* in Figure 5.16 has been aptly named. *Semimodularity* is a well-known paradigm for designing self-timed digital circuits without hazards

[7]See Chapter 3.7 for semimodularity.

by insisting that digital signal changes occur before being disabled. One might call it the "no change left behind" paradigm. Introduced by David Muller [29] and brought to the attention of a wider audience through Raymond Miller's 1965 book [27] semimodularity formed the starting point of the first generation of self-timed circuit design tools [19, 25]. Though semimodularity is still an important paradigm for designing and verifying self-timed circuits, new design trends for fast, energy-efficient self-timed circuits [4, 42, 44, 51] force it to share that position with *Relative Timing* [50]. The NuSMV code in lines 14–18 of Figure 5.16 for updating the variable *semimodular* is based on a new definition of semimodularity for timing constrained self-timed circuits, which was explained in Chapter 3.8.3 and presented in [32].

The module definition of *ff_posedge* in lines 22–31 of Figure 5.16 models a positive edge-triggered flipflop. The flipflop copies and stores the value of its second parameter, *d*, onto its output, *q*, whenever its first parameter, *ck*, changes from low (FALSE) to high (TRUE), as indicated in line 29. The value of output *q* is initialized through the third parameter, *init_q* (line 26). Instances of *ff_posedge* run each execution step, as indicated by the keyword *TRANS* in line 27.

To time and verify each *ff_posedge* instance, we pair it with a *process cgate* instance as its clock buffer. The clock buffer provides the timing flexibility in selecting when the flipflop acts. We verify the semimodular behavior of the clock buffer to ensure that all "clock" transitions issued by the *and2* gate reach the flipflop — see Section 5.2.1 for a reminder on "clocking." This explains the extra buffer *buf_ck* in Figure 5.14: it is the clock buffer for flipflop *FF*.

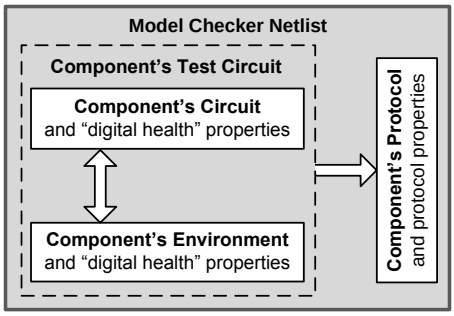
Gate models *cgate* and *ff_posedge* in Figure 5.15–5.16 have NuSMV code descriptions reminiscent of code descriptions in a hardware description language

like Verilog. We chose to use a general gate model for *cgate*, capable of modeling all combinational gates in the Click Storage component. This is possible because each gate instantiated in the component's gate-level netlist in Figure 5.14 has a behavioral description of its Boolean logic function. When instantiated with the signals coming into the gate, this Boolean logic function becomes the *set* function of the corresponding *cgate* instance in Figure 5.15. One could follow a similar approach for sequential gates and define a general gate model capable of modeling all sequential gates, as is done in [1]. We refrained from doing this here because Click components use only one type of sequential gate — a positive edge-triggered flipflop. Instead of using a few general gate models, one could define a dedicated model for each gate with a different logic function, and connect the gates by connecting their signal names. This is done in [9]. Figure 5.15 would require eight such dedicated gate models: two for the lazy environment, and six for the circuit. Dedicated gate models produce a larger Model Checker Library to characterize, but they contain extra connectivity information that could be useful.

5.2.2.3 Instantiating the Models in a Model Checker Netlist

Figure 5.17 repeats the middle grey rectangle of Figure 5.12 with the Model Checker Netlist but omits the white rectangle for the Component's Timing Constraints. It also shows the NuSMV translation with a single protocol, circuit, and environment instance for each. The keywords *process* in lines 4–5 indicate that the model checker will run the circuit and environment instances in asynchronous mode by interleaving their events. The *FAIRNESS running* command in line 11 insists that the event selection between the two instances be fair. The lack of keyword *process* in line 3 indicates that the protocol instance runs in synchronous

mode. This matches the modes of operation specified earlier in Figure 5.12.



(a) Model checker Netlist from Figure 5.12, but without timing constraints

```
1  MODULE main
2    VAR
3      ComponentProtocol     : protocol (in1_R, in1_A, out1_R, out1_A);
4      ComponentCircuit      : process circuit (in1_R, out1_A);
5      ComponentEnvironment  : process environment (in1_A, out1_R);
6    DEFINE
7      in1_R     := ComponentEnvironment.in1_R;
8      in1_A     := ComponentCircuit.in1_A;
9      out1_R    := ComponentCircuit.out1_R;
10     out1_A    := ComponentEnvironment.out1_A;
11   FAIRNESS running
```

(b) NuSMV code – instantiating component's protocol, circuit, and environment

Figure 5.17: Model Checker Netlist from Figure 5.12 without timing constraints and corresponding NuSMV code with one instance each of the component's protocol, circuit, and environment. The code of each instance can be found in Figures 5.13 and 5.15.

In Section 5.2.3, we will verify the "digital health" and protocol properties in the code of Figure 5.17, analyze any failing properties, and generate timing constraints to correct the failures. In Section 5.2.4 we will revisit Figure 5.17 and upgrade its NuSMV code by adding the missing constraints.

5.2.3 ARCtimer Step 3 — Timing Patterns

When the model checker runs the code in Figure 5.17 it reports multiple failing properties. For each failing property it gives a counterexample — a computation path that fails that property. Failing properties expose delay sensitivities in the design. A counterexample not only exposes a delay sensitivity, but also contains

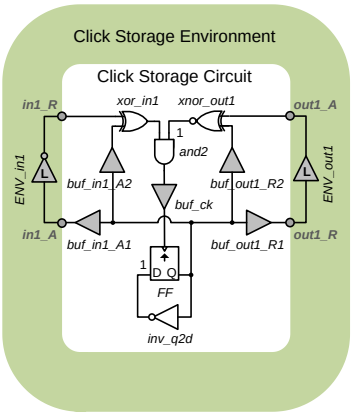
"clues" about how to prevent it from becoming hazardous. These clues can be captured in a form suitable for verification and correction — and thus prevention.

There are various options available for capturing clues. For instance, [64] assigns metric delay bounds to each gate in the circuit and its environment, capturing each clue as a tighter metric delay bound and calls this a *timing constraint*. Alternatively, a clue can be captured as a relative ordering of events and be called a *chain constraint* as in [30, 31], or a *(relative) timing constraint* as in [8, 18, 34, 52, 62].

Here, we capture a clue as a relative ordering of events and call this a *relative timing constraint*, or simply *constraint*.

Analyzing a counterexample to capture the clue it contains always requires finite state machine analysis around the failing step. Many of the approaches referenced here, notably [8, 18, 62–64], provide heuristics to capture the clue as a constraint and to generate the constraint automatically. These heuristics are, alas, tightly coupled to the underlying tools and theory and thus hard to transfer to other verification flows.

To share understanding of what is involved in analyzing a counterexample, we analyze the two counterexamples of Figure 5.18 for the netlist of Figure 5.17a — one failing a "digital health" property and the other failing a protocol property.



(a)

"Digital Health" Failure

Initial State:
state=s0
Run step 1:
ENV_in1.val+
in1_R+
state=s1
Run step 2:
xor_in1.val+
Run step 3:
and2.val+
Run step 4:
buf_ck.val+
FF.q+
Run step 5:
buf_out1_R1.val+
out1_R+
state=s3
Run step 6:
buf_out1_R2.val+
Run step 7:
ENV_out1.val+
out1_A+
state=s4
xnor_out1.semimodular=FALSE

Failure: CTLSPEC AG semimodular

Protocol Failure

Initial state:
state=s0
Run step 1:
ENV_in1.val+
in1_R+
state=s1
Run step 2:
xor_in1.val+
Run step 3:
and2.val+
Run step 4:
buf_ck.val+
FF.q+
Run step 5:
buf_out1_R1.val+
out1_R+
state=s3
Run step 6:
buf_out1_R2.val+
Run step 7:
xnor_out1.val−
Run step 8:
and2.val−
Run step 9:
buf_ck.val−
Run step 10:
inv_q2d−
Run step 11:
ENV_out1.val+
out1_A+
state=s4
Run step 12:
xnor_out1.val+
Run step 13:
and2.val+
Run step 14:
buf_ck.val+
FF.q−
Run step 15:
buf_out1_R1.val−
out1_R−
state=errorOUT

Failure: CTLSPEC AG state!=errorOUT

(b)

Figure 5.18: (a) Copy of the Click Storage circuit and environment coded in Figures 5.15–5.16 and two counterexamples (b) showing a "digital health" failure for gate *xnor_out1* and a failure in state *s4* of the protocol (Figure 5.13) monitoring the circuit and environment. The counterexamples each describe a path of events through the circuit. An event is a rising or falling signal transition. The counterexamples indicate rising signal transitions by appending the symbol "+" to the signal name, like ENV_in1.val+ in step 1. They indicate falling transitions by appending the symbol "−," like out1_R− in step 15 of the path with the protocol failure.

It is well to remember that the generation of relative timing constraints comes early in the design process, as part of building the library of handshake components — the Design Library in Figure 5.7 (left-column). Once constraints are known and stored in the Design Library, they are used over and over again for every chip design. Thus, the time taken for constraint generation plays only a small role in the overall time from design to market. We therefore have the leisure to make the constraints understandable to the component's designer, and to increase their robustness to circuit modifications applied later in the design process. We do this by formulating the constraints as *timing patterns*, in support of the *design patterns* that the designer selected for the component's circuit and family. The highly general and highly robust timing patterns derived for simple components can form a starter set for *priming* complex components. More detail on timing patterns appears in Section 5.2.3.3.

5.2.3.1 Analyzing Counterexamples

Figure 5.18b shows two counterexamples for the NuSMV netlist in Figure 5.17. To ease following the paths in each counterexample, Figure 5.18a repeats the gate-level circuit diagram of the Click Storage circuit and environment. Both counterexamples describe a path of events starting from the initial state. State names, like *s0* for the initial state, are filled in by the component's protocol — the vertical rectangle in the netlist diagram of Figure 5.17a. The protocol description for the Click Storage component can be found in Figure 5.13.

The two counterexamples show that if gates and wires have arbitrary delays it is harder to guarantee the simple operational descriptions of handshake components and handshake channel interfaces given in Section 5.2.1 and Figure 5.8.

The first six execution steps, run steps 1–6 in Figure 5.18b, are the same in both counterexamples. In run step 1, *ENV_in1* raises *ENV_in1.val*. The rising transition is denoted by the symbol "+" at the end of *ENV_in1.val*. For a falling transition we would have used the symbol "−." Remember that a gate name with suffix ".val" denotes the gate's output — see Figure 5.15–Figure 5.16. Because *ENV_in1.val* is an alternative name for *in1_R*, this run step changes the protocol state to *s1*. With *in1_R* high and *in1_A* still low, incoming channel *in1* is now full. This is detected by gate *xor_in* whose output rises in run step 2. With both its input signals high, AND function *and2* now "acts" as follows. First *and2.val* rises (run step 3), and then clock buffer output *buf_ck.val* rises and clocks flipflop *FF*, causing its output *FF.q* to rise (run step 4). From here on, the ordering of execution steps depends on the delays of the logic gate in the feedback loops from *FF.q* back to *buf_ck*. There are four such feedback loops — two per channel, on each side.

Both counterexamples focus on the two feedback loops at the *out1* side. They show what happens when the two feedback loops are equally fast, and what happens when both are faster than the two feedback loops at the *in1* side. The examples both select to change *buf_out1_R1* in run step 5, raising *buf_out1_R1.val* and thus *out1_R*, which changes the protocol state to *s3*. Outgoing channel *out1* is now empty. In run step 6, both examples then raise *buf_out1_R2.val*, making gate *xnor_out1* aware that *out1* is empty by enabling *xnor_out1.val* to fall. In run step 7, the two counterexamples diverge.

The example in the left box of Figure 5.18b selects *ENV_out1*, raising its output and thus *out1_A*, which changes the protocol state to *s4*. The change in *out1_A* also makes channel *out1* full and prevents *xnor_out1.val* from falling

before it took the opportunity to fall. This causes *xnor_out1.semimodular* to become FALSE (lines 15–16 of Figure 5.16) which is flagged because the gate has failed the "digital health" property, called *CTLSPEC AG semimodular* (line 20 of Figure 5.16).

A semimodularity failure like this could happen in a chip design if the internal path through the circuit, from *FF.q* via *buf_out1_R2* to *xnor_out1*, were to take about the same time as the external path through the environment, from *FF.q* via *buf_out1_R1* to *xnor_out1*. Were this to happen, it would render exclusive-NOR gate *xnor_out1* useless as a detector of full and empty channel states, thus defeating the handshake protocol on *out1*. To differentiate a full from an empty channel, *xnor_out1* must have enough time to receive and respond to the internal representation for *out1_R*, as captured by *buf_out1_R2*, before the environment responds with a next state change through *out1_A*. This is the clue we are seeking. Given that both inputs for *buf_out1_R1* and *buf_out1_R2* start at FF.q, or even at the AND function *and2.val* before that, we can capture this clue in the counterexample in one of the following two ways:

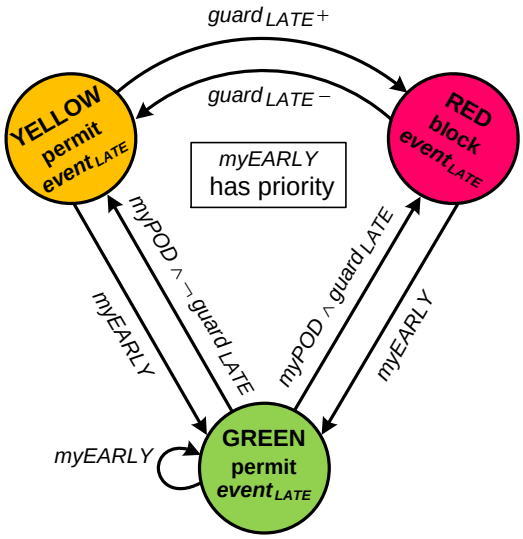
- After *FF.q* rises, *xnor_out1.val* must fall before *out1_A* rises. Following the notation of [9,52] we denote this as: $FF.q+ \rightarrow xnor\_out1.val- < out1\_A+$.
- If $\neg FF.q$ holds while *and2.val* rises, then subsequently *xnor_out1.val* must fall before *out1_A* rises — denoted as: $(\neg FF.q \wedge and2.val+) \rightarrow xnor\_out1.val- < out1\_A+$.

The second formulation of the captured clue matches relative timing constraint *rt3* in Figure 5.19. A similar counterexample exists for the case that *FF.q* holds while *and2.val* rises, leading to *rt4*. Two more such counterexamples can be found

by exchanging the two feedback loops at channel *out1*'s side for the two feedback loops at channel *in1*'s side. The four relative timing constraints *rt1* to *rt4* in Figure 5.19 block all such counterexamples.

Semimodularity failures are easy to solve: instead of disabling the transition, take it! This simple heuristic, however, tends to push the semimodularity failure to the next gate, just as a snow plow pushes snow elsewhere. This happens for instance between *rt7*–*rt8* and *rt9* in Figure 5.19, each of which solve a semimodularity failure. Constraints *rt7* and *rt8* solve a semimodularity failure for gate *and2* by pushing the failure to the next gate, *buf_ck*. Constraint *rt9* solves the semimodularity failure for gate *buf_ck* by pushing the failure to *FF*, which does not register this type of failure, and so the simple heuristic snow plow stops here. Relative timing constraints that merely push a semimodularity failure elsewhere fail to be appealing and intuitive to the designer of the component and are less robust to circuit modification applied later in the design process. We will come back to this when we discuss design patterns.

The counterexample in the right box of Figure 5.18 avoids the mistake of the first counterexample by taking the still-enabled transition *xnor_out1.val*− (run step 7). It continues by resetting the AND function and setting up the flipflop for the next handshake coordination (run steps 8–10). So far so good. But then, it starts a second handshake on channel *out1* (run steps 11-12) while ignoring the still outstanding first handshake on *in1* — forgetting that it "takes two to tango." With *in1_R* high and *in1_A* still low, input channel *in1* is still full and *xor_in1* is still high. As a result, the AND function "acts" prematurely (run steps 13–15) and coordinates the first handshake on *in1* with the second handshake on *out1*. This premature action of the AND function causes a protocol failure in run step 15.



(a) Stoplight model of a relative timing

Initial Set of Relative Timing Constraints for the Click Storage Circuit and Environment

myName		*myPOD*		*myEARLY*		*myLATE*
rt1	:	$(\neg FF.q \wedge and2.val+)$	$\rightarrow$	$xor\_in1.val-$	$<$	$in1\_R-$
rt2	:	$(\overline{FF.q} \wedge and2.val+)$	$\rightarrow$	$xor\_in1.val-$	$<$	$in1\_R+$
rt3	:	$(\neg\overline{FF.q} \wedge and2.val+)$	$\rightarrow$	$xnor\_out1.val-$	$<$	$out1\_A+$
rt4	:	$(\overline{FF.q} \wedge and2.val+)$	$\rightarrow$	$xnor\_out1.val-$	$<$	$out1\_A-$
rt5	:	$\overline{and2.val}+$	$\rightarrow$	$xor\_in1.val-$	$<$	$xnor\_out1.val+$
rt6	:	$and2.val+$	$\rightarrow$	$xnor\_out1.val-$	$<$	$xor\_in1.val+$
rt7	:	$and2.val+$	$\rightarrow$	$and2.val-$	$<$	$(\overline{xor\_in1.val} \wedge xnor\_out1.val+)$
rt8	:	$and2.val+$	$\rightarrow$	$and2.val-$	$<$	$(\overline{xnor\_out1.val} \wedge xor\_in1.val+)$
rt9	:	$and2.val+$	$\rightarrow$	$buf\_ck.val-$	$<$	$\overline{and2.val}+$
rt10	:	$FF.q+$	$\rightarrow$	$inv\_q2d.val-$	$<$	$buf\_ck.val+$
rt11	:	$FF.q-$	$\rightarrow$	$inv\_q2d.val+$	$<$	$buf\_ck.val+$

(b) Initial set of relative timing constraints

Figure 5.19: Initial set of relative timing constraints for the Click Storage component are derived by failure analysis of counterexamples for the NuSMV netlist in Figure 5.17. Failure analysis of the two counterexamples from Figure 5.18 in Section 5.2.3.1 gave us *rt3* and *rt5* — and implicitly *rt1* to *rt6*. Constraint $myNAME : myPOD \rightarrow myEARLY < myLATE$ expresses that after *myPOD* the computation encounters *myEARLY* before *myLATE*. The expressions *myPOD*, *myEARLY*, and *myLATE* formulate guarded events, as explained in the text on modeling relative timing constraints. The expressions for the guards are underlined. Rising events end with the symbol "+" and falling events end with the symbol "−."

This second counterexample violates the core purpose of the Click Storage component, which is to coordinate exactly one incoming handshake with exactly

one outgoing handshake and to repeat this for the successive handshakes on each channel. For one-to-one coordination, the AND function must know when a channel is willing to participate ("Shall we dance?") as well as when its participation is over ("Thank you!"). Each channel indicates its willingness to participate by raising the output of its exclusive-(N)OR gate, and each channel ends its participation by lowering this same output. After each action, both outputs must fall before either rises again. We capture this clue in the counterexample in the following way:

- When *and2.val* rises, then *xor_in1.val* must fall before *xnor_out1.val* rises. We denote this as:
  $and2.val+ \rightarrow xor\_in1.val- < xnor\_out1.val+$.

This formulation of the captured clue matches *rt5* in Figure 5.19. The related constraint, *rt6*, avoids similar counterexamples for the reverse situation by preventing each handshake on *in1* from outpacing its handshake partner on *out1*.

Solving protocol failures may be hard and require rules of thumb for designing self-timed circuits. For example [8] experiments with slow versus fast input events to guide the synthesis of self-timed circuits. By presuming a slow environment, it may be possible to generate *rt5* automatically from the second counterexample. We will come back to this when we discuss timing patterns.

5.2.3.2 Modeling Relative Timing Constraints

The relative timing constraints in Figure 5.19 capture the clues from the various counterexamples generated by the model checker. The two counterexamples of Figure 5.18 gave us *rt3* and *rt5*, and implicitly all six constraints, *rt1* to *rt6*. The

focus of the current section is to expose the structure and operation of all such relative timing constraints.

As mentioned in Section 5.1.4, relative timing constraints are constraints between signals at the ends of paths that start at the same point — signals that must change in a pre-established sequence. Each relative timing constraint identifies the point where the paths split, called a *Point of Divergence* (POD) in [9, 52] — here we call it *myPOD*. Each constraint also indicates the two destinations, a pre-established "early" end point and a pre-established "late" end point — we call these *myEARLY* and *myLATE*, respectively. In addition the constraint has a name, like *rt1* in Figure 5.19 — we call this *myNAME*.

Our relative timing constraints have the following structure:

- $myNAME : myPOD \rightarrow myEARLY < myLATE$

where

- *myPOD* is an abbreviation for: $\underline{guard_{POD}} \wedge event_{POD}$
  - $\underline{guard_{POD}}$ is a guard, i.e. a Boolean logic expression
  - $event_{POD}$ is an event, i.e. a rising or falling signal
- *myPOD* holds if and only if
  - $event_{POD}$ occurs, and
  - meanwhile $\underline{guard_{POD}}$ holds
- *myEARLY* and *myLATE* have similar structures:
  - *myEARLY* abbreviates $\underline{guard_{EARLY}} \wedge event_{EARLY}$
  - *myLATE* abbreviates $\underline{guard_{LATE}} \wedge event_{LATE}$

To better distinguish guards from events, we underline guards. We omit trivial guards, like TRUE. For instance, the guards for *myPOD* in *rt5* to *rt11* are omitted for this reason.

Constraint $myNAME\colon myPOD \rightarrow myEARLY < myLATE$ says:

- if *myPOD* becomes valid
- then *myEARLY* must become valid
- before *myLATE* becomes valid.

One can use a constraint for analysis and report whether or not it is satisfied for all possible computation paths of the system. This is done, for instance, during static timing analysis — see Section 5.1.4. Alternatively, one can use a constraint as an actuator — a delay device that retards $event_{LATE}$ after *myPOD* becomes valid by blocking $event_{LATE}$ until *myEARLY* has become valid. The model checker uses constraints as actuators.

Our model checker's actuator model of constraint *myNAME* is a three-state finite state machine extension of the two-state version used in [9] and in Section 4.1.3. The three states are necessary for modeling the non-trivial guards of *myLATE* in *rt7* and *rt8* of Figure 5.19. We name the three states GREEN, YELLOW, and RED.

Figure 5.19a shows the *stoplight* model that we use as the model checker's actuator view of a relative timing constraint. Both GREEN and YELLOW states permit $event_{LATE}$ to happen, while a RED state blocks $event_{LATE}$. Most constraints start in GREEN, as do *rt1* to *rt11* in Figure 5.19b, and proceed as follows:

- All constraints go to a GREEN state when *myEARLY* becomes valid, because the need to retard $event_{LATE}$ vanishes with arrival of *myEARLY*.

- In GREEN, only *myPOD* can change the state, because *myEARLY* and *myLATE* matter only after *myPOD* becomes valid. The stoplight changes from GREEN to YELLOW if *myPOD* holds but $guard_{LATE}$ does not. Only instances of $event_{LATE}$ for which $guard_{LATE}$ holds need blocking. The state changes from GREEN to RED if both *myPOD* and $guard_{LATE}$ hold.

- Both YELLOW and RED states follow from arrival of a valid *myPOD* but not yet a valid *myEARLY*.

- Before *myEARLY* becomes valid, changes in $guard_{LATE}$ change the state from YELLOW to RED, and vice versa. Such changing of the guard and the state happens in some computations for *rt7* and *rt8* in Figure 5.19b. The stoplight state for *rt7* or *rt8* changes from RED, at *myPOD*, to YELLOW, by *rt5* and *rt6*, back to RED if *xor_ in1.val+* or *xnor_ out1.val+* changes before *and2.val−*.

5.2.3.3 Deriving Timing Patterns

Failure analysis of the two counterexamples in Figure 5.18 gave us the first six relative timing constraints *rt1* to *rt6* of Figure 5.19. Constraints *rt1* to *rt6* are the weakest relative timing constraints required to prevent the failures exposed by the two counterexamples and similar examples. The remaining constraints are also the weakest relative timing constraints of their kind:

- Constraints *rt7* and *rt8* form the weakest relative timing constraints to maintain the "digital health" of gate *and2* as a semimodular gate. They go as far as to permit a single rising *and2* input after both inputs have gone low, before they require *and2.val* to fall.

- Constraint *rt9* is the weakest relative timing constraint to maintain *buf_ck*'s "digital health" as a semimodular gate.

- Constraints *rt10* and *rt11* are the weakest setup time constraints for positive edge-triggered flipflop *FF*.

One can explore $myNAME\colon myPOD \rightarrow myEARLY < myLATE$ expressions to get an idea which constraints are critical. One way to do this is to estimate the elapsed time between *myLATE* and *myEARLY* at full speed operation under reasonable gate delays and in a reasonable environment, e.g.:

- Assume gate delays equivalent to 2 inverter delays for X(N)OR, AND, FF. Assume zero delay for the grey buffers. Replace the component's environment in Figure 5.15 by two other Click Storage circuits, one on each channel. Assume maximally parallel operation — no stalling.

- Under these assumptions, the cycle time for *and2.val+* is 12 inverter delays, and the elapsed time from *myEARLY* to *myLATE* is 4 inverter delays for *rt1–rt4* and *rt7–rt8*, 6 for *rt5–rt6* and *rt9*, and 9 for *rt10–rt11*.

With at least 4 inverter delays to spare in each constraint, these estimations indicate that the risk for violating a constraint is low and that none of the constraints *rt1* to *rt11* is critical.

Constraints *rt1* to *rt11* are the weakest possible constraints in part because they are tightly coupled to the circuit. A tight coupling between constraints and circuit is useful if the chip uses exactly this circuit for each instance of the Click Storage component — which is unlikely. For example, a technology mapping tool might partition the AND gate into a NAND and inverter, and a layout tool might

add clock buffers. With a NAND gate or extra clock buffers, constraints *rt1* to *rt11*, as formulated in Figure 5.19, no longer suffice because the gate names and connections have changed. To make the constraints suffice might require a grouping of gates in the new circuit and a mapping of group names back to the old circuit. This is common practice and not a problem in itself. The problem is that not all renamings ensure that *rt1* to *rt11* still cover all the properties in the new circuit — see Figure 5.20.

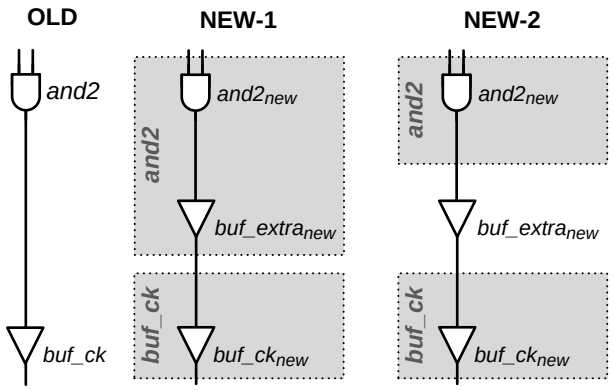


Figure 5.20: (OLD) Click Storage sub-circuit from Figure 5.15, and (NEW) two new post-layout versions with an extra buffer. Sub-circuit NEW-1 groups $and2_{new}$ and buf_extra_{new} and identifies the group with *and2* in OLD. The semimodular gate behavior of each gate is covered if the semimodularity of *and2* is covered. Constraints *rt7* and *rt8* of Figure 5.19 cover the semimodularity of *and2*, and thus that of $and2_{new}$ and buf_extra_{new} in NEW-1. The grouping and renamings for sub-circuit NEW-2, however, keep buf_extra_{new} isolated. Because *rt1* to *rt11* are the weakest possible constraints for the original circuit with sub-circuit OLD, they fail to cover the semimodular gate behavior of buf_extra_{new} in NEW-2. Because the timing patterns *p5* and *p6* in Figure 5.21 cover the semimodular behavior of all gates from *and2* through *buf_ck*, they cover the semimodular behavior of the intermediary gate, buf_extra_{new}, in both NEW-1 and NEW-2.

Ensuring that the renaming works for *rt1* to *rt11* may require re-running the model checker on the new circuit. However, re-running the model checker would defeat the purpose of working with a Design Library of verified components and would put the timing verification framework, i.e. ARCtimer, in the critical design

cycle of each chip. Our purpose holds to keep ARCtimer firmly in the early part of the design process.

To hold this purpose, the constraints must work regardless of circuit changes made during technology mapping or layout. Figure 5.20 shows that constraints *rt1* to *rt11* fail this purpose.

In summary: *We need general constraints that emphasize the circuit's intent rather than the circuit's structure.*

The component's designer faces a similar issue when choosing appropriate structures for the component's circuit. To make the circuit work for every chip, he or she uses design patterns. The patterns work for most technology mappings and layout tools. We wish to solve circuit design and circuit timing in a similar way. We seek *timing patterns* that make the *design patterns* work — i.e. that ensure:

- the X(N)OR gates detect full and empty channel states,
- the AND function coordinates the handshakes, and
- the *FF* and *inv_q2d* flip the channel state.

Let us examine the initial constraints *rt1* to *rt11* of Figure 5.19 to see which might work as patterns and which need generalizing:

- Constraints *rt1* to *rt4* make the X(N)OR gates work, and do no more and no less than that — they make fine patterns. Figure 5.21 rephrases them as *p1* and *p2*.
- Constraints *rt5* and *rt6* make the AND function work by comparing the output signals of the X(N)OR gates. This comparison makes less sense for complex AND functions in components with more than one channel on each

side. Requiring the outputs of all X(N)OR gates to fall before any channel input changes results in the more general constraints *p3* and *p4* in Figure 5.21 (top) and *p* (bottom).

- Constraints *rt7* to *rt9* keep the AND function semimodular, but they do this by exposing the organization of the AND function all the way from gate *and2* to the *FF*'s clock input — a result of resolving semimodularity failures by pushing them out of the way. The slow environment presumed in Section 5.2.3.1 for *rt5* and *rt6* can be assumed again here to guarantee that there will be enough time to stabilize internal feedback loops up to *FF*'s clock input before the channel inputs change. This assumption is formalized in *p5* and *p6* of Figure 5.21. Unlike *rt7* and *rt8*, patterns *p5* and *p6* are robust to both post-layout design changes shown in Figure 5.20.

- Constraints *rt10* and *rt11* keep the *FF* with *inv_q2d* combination flipping, but can be generalized as patterns *p7* and *p8* of Figure 5.21 by assuming a slow environment.

Note that each pattern in *p1* to *p8* of Figure 5.21 still leaves at least 2 inverter delays to spare under the earlier estimations for full speed operation, reasonable gate delays, and a reasonable environment. This indicates that the risk for violating one of these patterns is low and that none of them is critical.

The slow environment assumed above leads to a *burst-mode* operation [12] of the Click Storage component, where internal loops stabilize before new external channel inputs arrive. The burst-mode assumption is expressed most clearly in the parametrized pattern *p* of Figure 5.21. It is quite common in self-timed circuit design to assume that an external feedback loop through the component's

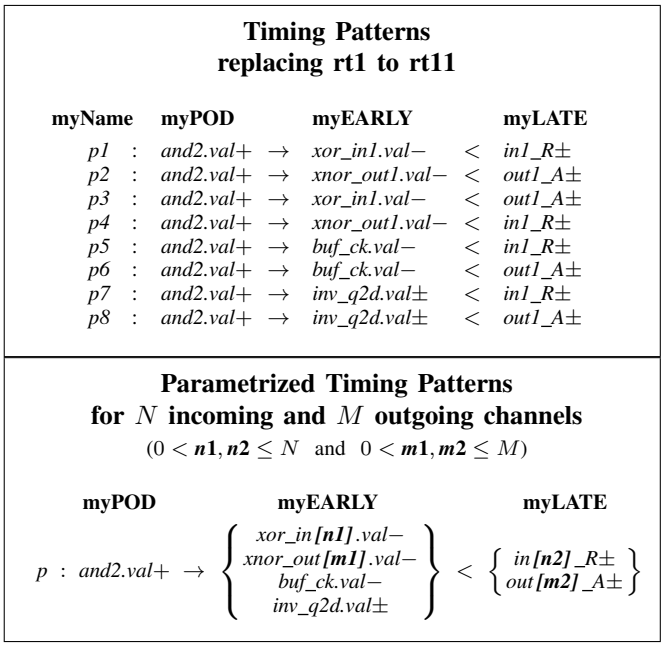


Figure 5.21: Timing Patterns for a Click Storage component with a single incoming and a single outgoing channel (top). Pattern p (bottom) re-phrases and parametrizes *p1* to *p8* to multiple incoming and outgoing channels. It expresses that after *myPOD* the computation must satisfy all *myEARLY* before any *myLATE*. Symbols "+," "−" and "± " at the end of a signal indicate a rising, falling, or either signal transition.

environment is slow compared to an internal feedback loop in the component's circuit. Heuristics for automatic circuit synthesis or timing constraint generation often use such assumptions. There is no guarantee, however, that relative timing constraints generated on the basis of heuristics are sufficiently general to be stored in a Design Library for use in every chip design.

The role of ARCtimer's Step 3 is to take the initial timing constraints, obtained by human or automated failure analysis, and turn them into sufficiently general timing patterns.

5.2.4 Step 2 Revisited — Adding Timing Constraints

The double-headed arrow in Figure 5.7 (right-column), on the spiral between Step 2 and Step 3, indicates that we alternate these two steps. We first run the model checker (Step 2), then we examine a few counterexamples and capture their clues in one ore more relative timing constraints (Step 3). Then we model the constraints, and re-run the model checker primed with these constraints. We examine a few counterexamples, and repeat. We alternate Step 2 and Step 3 until the model checker reports no further counterexamples. This alternation gave us constraints *rt1* to *rt11* of Figure 5.19b, from which we then derived timing patterns *p1* to *p8* and *p* of Figure 5.21.

The purpose of this Section is to illustrate how one can model such constraints in a general-purpose model checker. As before, we use the NuSMV model checker as example. Figure 5.22 expands the NuSMV Model Checker Netlist of Figure 5.17a without timing constraints to match the version shown in Figure 5.12 (middle) with *p1* to *p8* as the component's timing constraints. When the model checker runs the code in Figure 5.22 it reports no further counterexamples.

```
MODULE rt (eventPOD, eventEARLY, init_rt, guardPOD, guardEARLY, guardLATE, xPOD, xEARLY)
   VAR
      stoplight : {GREEN, YELLOW, RED};
   ASSIGN
      init(stoplight) := init_rt;
   TRANS
      next(stoplight) = case
         myEARLY                                  : GREEN;
         stoplight=GREEN  & myPOD & next(!guardLATE): YELLOW;
         stoplight=GREEN  & myPOD & next(guardLATE) : RED;
         stoplight=YELLOW & next(guardLATE)         : RED;
         stoplight=RED    & next(!guardLATE)        : YELLOW;
         TRUE                                     : stoplight;
      esac;
   DEFINE
      myPOD   := guardPOD & ((xPOD & eventPOD!=next(eventPOD)) | (!eventPOD & next(eventPOD)));
      myEARLY := guardEARLY & ((xEARLY & eventEARLY!=next(eventEARLY)) | (!eventEARLY & next(eventEARLY)));
      stop    := (stoplight=RED);
   --PROPERTIES
   --safety
      CTLSPEC AG (stoplight=YELLOW -> !guardLATE) & (stoplight=RED -> guardLATE)
--END MODULE RTconstraint

MODULE circuit (in1_R, out1_A)
  --RELATIVE TIMING CONSTRAINTS
  VAR
    p1p3: rt (and.val, !xor_in.val,   GREEN, t,t,t,f,f);
    p2p4: rt (and.val, !xnor_out.val, GREEN, t,t,t,f,f);
    p5p6: rt (and.val, !ckbuf.val,    GREEN, t,t,t,f,f);
    p7p8: rt (and.val, inv_q2d.val,   GREEN, t,t,t,f,t);
  DEFINE
   stop_in1_R_x   := p1p3.stop | p2p4.stop | p5p6.stop | p7p8.stop;
   stop_out1_A_x  := p1p3.stop | p2p4.stop | p5p6.stop | p7p8.stop;
 VAR
    xor_in1     : process cgate (in1_R xor buf_in1_A2.val, f,f,f,f);
    xnor_out1   : process cgate (out1_A xnor buf_out1_R2.val, t,f,f,f);
    and2        : process cgate (xor_in1.val  & xnor_out1.val, f,f,f,f);
    buf_ck      : process cgate (and2.val, f,f,f,f);
    FF          : ff_posedge    (buf_ck.val, inv_q2d.val, f);
    inv_q2d     : process cgate (!FF.q, t,f,f,f);
    buf_in1_A1  : process cgate (FF.q, f,f,f,f);
    buf_in1_A2  : process cgate (FF.q, f,f,f,f);
    buf_out1_R1 : process cgate (FF.q, f,f,f,f);
    buf_out1_R2 : process cgate (FF.q, f,f,f,f);
  DEFINE
    in1_A       := buf_in1_A1.val;
    out1_R      := buf_out1_R1.val;
    f           := FALSE;
    t           := TRUE;
  FAIRNESS running

MODULE environment (in1_A, out1_R, stop_in1_R_x, stop_out1_A_x)
  VAR
    ENV_in1    : process cgate (!in1_A, f, t, stop_in1_R_x,  stop_in1_R_x);
    ENV_out1   : process cgate (out1_R, f, t, stop_out1_A_x, stop_out1_A_x);
  DEFINE
    in1_R      := ENV_in1.val;
    out1_A     := ENV_out1.val;
    f          := FALSE;
    t          := TRUE;
  FAIRNESS running

MODULE main
  VAR
    ComponentEnvironment  : process environment (in1_A, out1_R);kill
    ComponentProtocol     : protocol (in1_R, in1_A, out1_R, out1_A);
    ComponentCircuit      : process circuit (in1_R, out1_A);
    ComponentEnvironment  : process environment (in1_A, out1_R, stop_in1_R_x, stop_out1_A_x);
  DEFINE
    stop_in1_R_x  := ComponentCircuit.stop_in1_R_x;
    stop_out1_A_x := ComponentCircuit.stop_out1_A_x;
    in1_R         := ComponentEnvironment.in1_R;
    in1_A         := ComponentCircuit.in1_A;
    out1_R        := ComponentCircuit.out1_R;
    out1_A        := ComponentEnvironment.out1_A;
  FAIRNESS running
```

Figure 5.22: NuSMV model checker code changes are applied for adding the relative timing constraints captured in patterns *p1* to *p8* of Figure 5.21. The Model Checker Library adds the module definitions for *protocol* (Figure 5.13) and logic gates *cgate* and *ff_posedge* (Figure 5.16). **Note:** In NuSMV, earlier commands in a case statement have a higher priority, and the symbol "!" is used for logical negation.

Module *rt* in lines 1–18 of Figure 5.22 models the state changes of the *stoplight* model of Figure 5.19a which is our model of a relative timing constraint. The parameters in line 1 with names *eventPOD*, *eventEARLY*, *guardPOD*, *guardEARLY*, and *guardLATE* represent respectively $event_{POD}$, $event_{EARLY}$, $guard_{POD}$, $guard_{EARLY}$ and $guard_{LATE}$ defined earlier, and used in Figure 5.19. Note the absence of a parameter for $event_{LATE}$. The third parameter in line 1, *init_rt*, contains the initial stoplight state. The role of the last two parameters, *xPOD* and *xEARLY*, is to reduce the number of *rt* instances needed to code constraints. Parameter *xPOD*, when TRUE, indicates that both rising and falling signal transitions count as events for *myPOD*. Parameter *xEARLY* indicates the same for *myEARLY*. The last two parameters make it possible to model *inv_q2d.val*$\pm$ in *p7* and *p8* of Figure 5.21 with a single *rt* instance by making *xEARLY* TRUE (t) — see line 26 in Figure 5.22.

The statement between the keywords *case* and *esac* in lines 7 and 14 is precisely the code for the stoplight's state changes. It is executed in synchronous mode, i.e. in each execution step by the model checker, as indicated by the keyword *TRANS* in line 6. This is consistent with the mode of execution indicated earlier in Figure 5.12 for the leftmost white rectangle with the name "Component's Timing Constraints."

The signal, *stop*, defined in line 18 of Figure 5.22, is TRUE if and and only if the stoplight is RED.

The constraint's *rt* instances follow in lines 25–33, in the code for the *circuit* module. Constraints that start in the same state and that use the same *myPOD*, *myEARLY*, and $guard_{LATE}$ have the same *rt* parameters in the NuSMV code, and can thus share an *rt* instance. For instance, patterns *p1* and *p3* share an *rt*

instance in line 27, called *p1p3*. This is possible, despite the fact that the two patterns have different *myLATE* events. It is possible because the *rt* instances control the GREEN, YELLOW and RED stoplight states, but they do not block *myLATE* event. The *cgate* instance that drives the *myLATE* event, $event_{LATE}$, is the one that blocks the event.

For the model checker, we partitioned the stoplight model of Figure 5.19a into a stoplight controller (*rt*) and a driver (*cgate*). Although [9] uses a two-state model instead of our three-state stoplight model, their model checker solution uses exactly the same partition. The analogy with everyday stoplights and drivers makes this an obvious partition.

Just as multiple stoplights may force a driver to stop a vehicle, so multiple *rt* instances may force a *cgate* to block a *myLATE* event. Take for instance relative timing constraints *p1*, *p4*, *p5*, and *p7* of Figure 5.21. The constraints share *myLATE* event, *in1_R*± , which corresponds to a rising or falling transition. This means that any of these constraints may block any transition of *in1_R*. Thus, in the model checker, each of the *stop* signals of the constraints' multiple *rt* instances may block *in1_R*± . Line 32 of Figure 5.22 combines these separate *stop* signals into a single variable *stop_in1_R_x*, to simplify the remaining code. Likewise, line 33 combines the separate *stop* signals for *myLATE* event *out1_A*± into a single variable *stop_out1_A_x*.

All events *in1_R*± and *out1_A*± are generated by the environment. Therefore, *stop_in1_R_x* and *stop_out1_A_x* must pass from the instantiated circuit to the instantiated environment through the parameter mechanism, as shown in lines 63–75 of Figure 5.22. In lines 52–60, module *environment* passes the parameters to the appropriate *cgate* drivers, *ENV_in1* and *ENV_out1*.

Because *stop_in1_R_x* blocks the rising as well as the falling transitions of *ENV_in1.val* it is passed to both the *stop_rise* and the *stop_fall* parameter slots for *cgate ENV_in1* — making the *cgate* block any transition of *ENV_in1.val* if *stop_in1_R_x* is TRUE (line 9 of Figure 5.16). This is how timing constraints control the run-time values of *stop_rise* and *stop_fall* in the various *cgate* drivers of *myLATE* events.

5.2.5 ARCtimer Step 4 — Static Timing Analysis

Section 5.2.3 ended Step 3 "Timing Patterns" of Figure 5.7 (right-column-bottom) with a set of timing patterns *p1* to *p8* and their parametrized version p — see Figure 5.21. This set is complete in terms of property coverage and sufficiently general to apply to every chip with a Click Storage component.

This Section 5.2.5 takes p to Step 4 "Static Timing Analysis" of Figure 5.7 (left-column-bottom) by translating p's formula into code for static timing analysis (STA). We store both p's formula, $p : myPOD \rightarrow myEARLY < myLATE$, and its STA code in the Design Library.[8]

The first task for static timing analysis is to validate p, i.e. to validate that p's slowest early path is faster than p's fastest late path in the chip's gate-level netlist. This involves computing the maximum path delay, max_{EARLY}, of all paths from *myPOD* to *myEARLY* and the minimum path delay, min_{LATE}, from *myPOD* to *myLATE*, and validating that:

- $max_{EARLY} < (min_{LATE} + margin)$

  for some delay *margin*.

[8]The STA code for non-relative-timing constraints, such as minimum clock pulse widths, can be stored, organized, and generated in a way similar to p.

The second task for static timing analysis is to repair the netlist in case the first task invalidates p. The iterative repair process described in Section 5.1.4 performs this second task. It finds the minimum delay value d to make p valid, given a delay insertion point in the netlist at which to insert d.

Calculating max_{EARLY} and min_{LATE} involves following the topological connections between gates and wires in the netlist, differentiating rising from falling transitions where possible, and filling in gate and wire delays using lookup tables [15, 26, 37, 38, 52, 61]. Unfortunately, some STA tools cannot differentiate rising from falling transitions, and many STA tools cut paths and loops at flipflops — see Section 5.1.4. As a result, most STA tools need guidance to know if a path passes through or bypasses a flipflop and to know which delay to use at asymmetric delay insertion points.

Various self-timed design groups have developed solutions to guide STA tools through a gate-level netlist with self-timed circuitry — see Section 5.1.4. The solutions usually involve pre-cut sub-paths that a conventional STA tool can handle. These pre-cut sub-paths are the result of a higher-level analysis of the netlist. The higher-level analysis is the most interesting part of any of these solutions, because it is the part that would remain necessary even if conventional STA tools were capable of doing the analysis without guidance.

The STA code stored in the Design Library does the higher-level analysis. It contains the algorithms to find paths and calculate path delays and to mark intermediary flipflops and other relevant checkpoints on the paths.[9] Below, we indicate the most important decisions that we made to organize this STA code. These decisions complement the actual path cutting pragmatics in [15, 26, 38, 61].

[9]We call these *checkpoints*, after the Berlin Wall's "Checkpoint Charlie" — the famous Cold War crossing point between East and West Berlin.

- *Fill in crucial semantic details in advance:*

  We use the model checker and formal analysis to fill in behavioral details that a topological search cannot find.

- *Mimic the modularity of the self-timed design:*

  Because the Design Library stores information by component, we must partition the STA code also by component. Our self-timed components communicate by handshakes over channels, as explained in Section 5.2.1 and Figure 5.8. Each handshake is marked by a pair of events, making the channel full and then empty, or vice versa. In Click these events are marked by a transition on the request signal followed by a transition on the acknowledge signal, or vice versa. Each pair of handshake events partitions the paths in the netlist between two successive components. Thus, although we store the main STA code for validating the component's timing constraints with the component, we can distribute the full code by storing the delay calculations for the other side of a partitioned path with the other component.[10] The STA code uses the pair of handshake events to initiate an external delay calculation and return its results. This process may be recursive, because the STA code for the neighboring component may initiate sub-calculations stored with further out neighboring components before it can complete its calculation. We implement this using *channel subroutine calls*. Hence, in addition to STA code for validating its own timing constraint, each component must also store STA code for the *channel subroutines* for which it might receive calls.

[10]Bundled data setup and hold time constraints use a similar partition based on different pairs of handshake events — between request and data signals versus acknowledge and data signals. See also footnote 1.

- *Sequence the calculations in a sensible way*

  Internal paths generally contribute less delay than paths that exit and enter the component via a handshake channel. Thus, it makes sense to start minimum path delay calculations with internal paths, and use the current minimum to cut off subsequent calculations including the channel subroutine calls introduced above.

As an example, let us look at the decisions and STA code organization related to timing constraint p of Figure 5.21 and its calculations to validate $max_{EARLY} < (min_{LATE} + margin)$:

- *Delay insertion points:*

  We have chosen to repair p at the two *myLATE* events, by delaying signal changes on *in[n2]_R* and *out[m2]_A* — whichever applies. The two end signals make good repair points, because, not only do they change exactly once per *myPOD–myLATE* cycle, the minimum frequency for repair, but also their change covers all of the *myEARLY* events in each *myPOD–myLATE* cycle. The delay element must delay both rising and falling transitions because the direction of the change is irrelevant, as indicated by the symbol "$\pm$ " in Figure 5.21.

  Also, as p's *myLATE* events, *in[n2]_R* and *out[m2]_A* share the same set of *myEARLY* events. As a result, we can delay signal changes on *in[n2]_R* and *out[m2]_A* without creating circular repair dependencies. The lack of circular dependencies ensures that the repair process described in Section 5.1.4 will converge.

- *Additional semantic details to simplify max_{EARLY}:*

  Timing constraint p has falling signal transitions for *myEARLY* events, and

thus requires transition-aware static timing analysis. However, the only *myEARLY* event preventing a transition-agnostic analysis is *buf_ck.val−*. Unlike its transition-agnostic version *buf_ck.val±* , event *buf_ck.val−* follows *and2.val+* not immediately but only after *FF.q±* . As it happens, all *p*'s *myEARLY* events follow *and2.val+* after *FF.q±* . We can indicate this by adding *FF.q±* between *myPOD* and *myEARLY* in *p*. The presence of *FF.q±* makes it possible to focus on changes rather than specific transitions of *myEARLY* events — the transitions are implied, as the model checker can confirm. Figure 5.23 shows the updated version of *p*, with checkpoint *FF.q±* and non-specific *myEARLY* event transitions. This is the version that we translate into STA code, using transition-agnostic calculations.[11]

From *p* itself we can deduce that early paths end before any *myLATE* event and thus never go through a channel. Therefore, we can use transition-agnostic calculations for max_{EARLY} restricted to paths internal to the module.

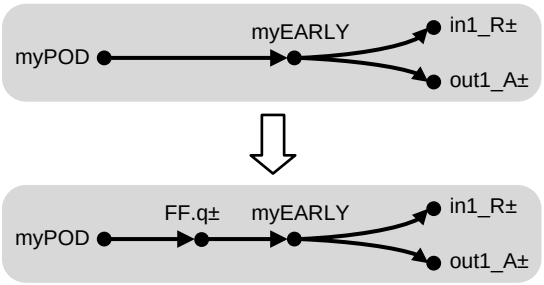
- *STA code partitioning for min_{LATE}:*

  The min_{LATE} STA code for p calculates the minimum path delay for paths from *myPOD* to *myLATE* internal to the module — no holds barred. The code keeps track of any flipflop or other checkpoints that may need further preparation before the calculations can be handed over to conventional STA

[11] We chose *and2.val+* as *p*'s *myPOD* rather than *FF.q±* , because as the AND function of the Click Storage component, *and2.val+* makes the component "act" more so than *FF.q±* . Moreover, alternative circuit implementations that split *FF* into separate flipflops for each channel [39] require *p* to use *and2.val+* as *myPOD*. Having said that, the Click Storage circuit in Figure 5.7 (right-column-top) can use *FF.q±* as *myPOD* and thus, without inserting an additional checkpoint, avoid the need to differentiate rising from falling transitions in *myEARLY*. Whether one chooses to keep the STA code as general as possible by taking *and2.val+* as *myPOD* or as simple as possible by taking *FF.q±* as *myPOD* the goal remains the same: simplify the STA code using transition-agnostic path-finding and path-delay calculations where possible.

tools. Each time the path subsequently exits and enters the module over a channel, the code inserts a channel subroutine call, and splits the calculation into the sum of three sub-calculations: the original calculation up to the exit over the channel, the channel subroutine call, and the original calculation from the entry over the channel back into the module.

Each channel comes with STA code to fill in the delay of a channel subroutine call entering and exiting the Click Storage module. Each such channel subroutine calculates the minimum path delay for paths through the component from channel entry to channel exit — no holds barred.



Parametrized Timing Patterns for STA code generation
replacing p (***guard*** $\stackrel{def}{=}$ *sta1.stoplight* $\neq$ GREEN)

myPOD		***myEARLY***		***myLATE***
sta1 : *and2.val+*	$\rightarrow$	*FF.q±*	$<$	$\left\{ \begin{array}{c} in\,\textbf{[n2]}\_R\pm \\ out\,\textbf{[m2]}\_A\pm \end{array} \right\}$
sta2 : (***guard*** $\wedge$ *FF.q±*) $\rightarrow$		$\left\{ \begin{array}{c} xor\_in\,\textbf{[n1]}.val\pm \\ xnor\_out\,\textbf{[m1]}.val\pm \\ buf\_ck.val \wedge buf\_ck.val\pm \\ inv\_q2d.val\pm \end{array} \right\}$	$<$	$\left\{ \begin{array}{c} in\,\textbf{[n2]}\_R\pm \\ out\,\textbf{[m2]}\_A\pm \end{array} \right\}$

Figure 5.23: We modify timing pattern p of Figure 5.21 to simplify its STA code. Knowing that all p's *myEARLY* paths go through flipflop *FF* compensates the need to differentiate rising from falling transitions in *myEARLY*. The graphs show the event orderings specified by p before (top) and after (bottom) adding *FF* as intermediary checkpoint. The constraints *sta1* and *sta2* shown in the box formulate the modified version of p, and have been verified by using their NuSMV translations instead of those of p in lines 27–33 of Figure 5.22. The guard in *myPOD* of *sta2* plays the role of the baton in a relay race, handing over the task of blocking *myLATE* from *sta1* to *sta2*. The guard in *myEARLY* for *buf_ck.val±* in *sta2* distinguishes the targeted early transition from the one occurring concurrently with *myPOD*, and can be ignored in the STA code.

Our STA code calculations are conservative. They tend to ignore any guards, and focus only on the event changes in the relative timing constraint formulation of Section 5.2.3.2. This is possible because we compensate for missing details by adding checkpoints. Moreover, static timing validation is more forgiving than behavior-based timing verification: it suffices to satisfy $max_{EARLY} < (min_{LATE} + margin)$ even were the minimum and maximum delays to belong to false paths.

5.2.6 Summary Timing Verification Framework Steps 1–4

Our Design Library, Figure 5.7 (left-column), stores a set of handshake components for use in larger self-timed systems. For each component, the Design Library stores a circuit description, a protocol description, a description of the timing constraints for the circuit, and static timing analysis code to validate and enforce these constraints in the final system. The circuit and its timing constraints are known to follow the protocol properly because they have gone through the verification steps outlined here. Because these verification steps happen early in the design process, we have the leisure to apply an in-depth verification process.

We make use of a model checker as part of the verification process. The model checker verifies that each component, or rather its timing constrained circuit, obeys the protocol specified for its interface signals. The model checker also verifies the "digital health" of a component. "Digital health" includes such properties as *semimodular gate behavior* and as *absence of set-reset drive fights*, a "digital health" property not used in this Chapter but important for verification of GasP components. The static timing analysis code covers additional timing constraints, such as minimum clock pulse widths for all the edge-triggered flipflops in the Click

circuit. The flipflop models that we used are too abstract for the model checker to detect the need for such pulse width constraints.

In building the Design Library, we strive for modularity and generality. The Design Library is organized by component. Even the static timing analysis code generated in Step 4 is partitioned over the components. For each handshake component we seek circuit descriptions as well as protocol, constraint, and code descriptions that are understandable to the component's designer, easy to maintain, and robust to circuit modifications applied later in the design process. Where possible, the descriptions in the Design Library are parametrized to address variable numbers of channels. Whenever we use the term *pattern*, as in *design pattern* or *timing pattern*, it is to emphasize the generality of that particular description.

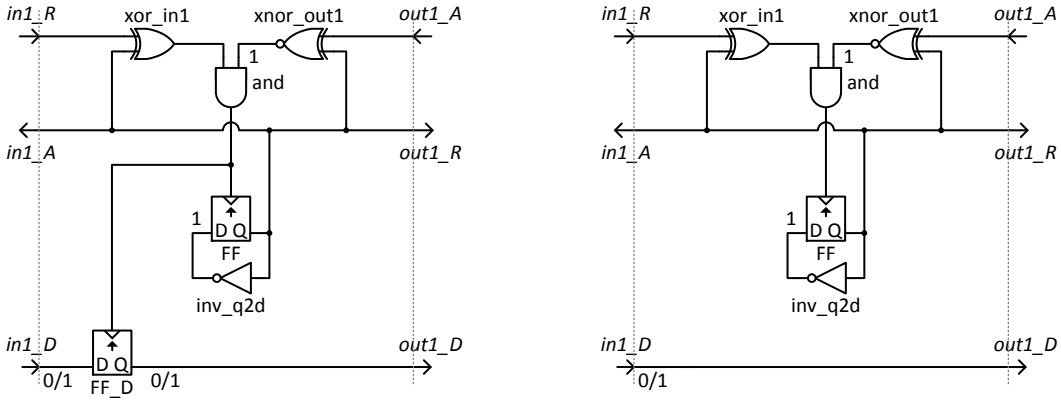
6

RT characterization of Bounded Bundled Data

This Chapter brings the datapath into the verification flow discussed in Chapter 5. We explain how data is modeled and characterize the RT constraints for a Click Storage component with *bounded-bundled data* (BBD) protocol.

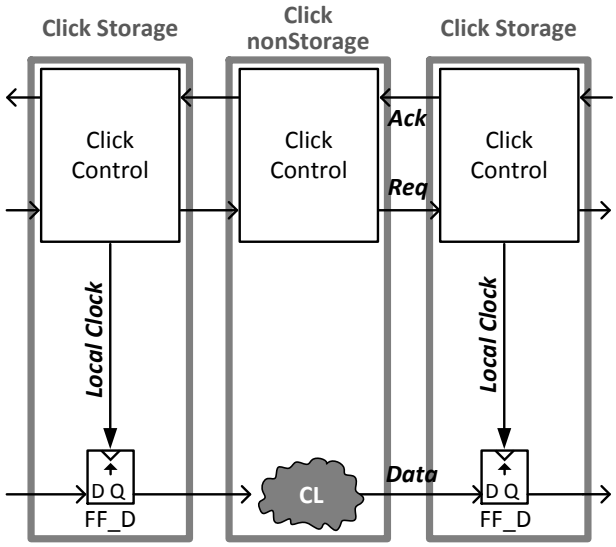
6.1 Bounded Bundled Data and Click Storage

Figure 6.1 shows two circuit implementations of Click storage with datapath. The first stores the data locally, the second doesn't store the data. When the predecessor on the input channel is full, the XOR gate *xor_ in1* becomes high. XNOR gate *xnor_ out1* is high when the output channel is empty, and low when the output channel is full. The AND gate locally synchronizes the predecessor's and successor's handshakes, clocks the flipflop in the controller *FF*, and also clocks the flipflop(s) in the datapath *FF_ D* if local data storage exists.

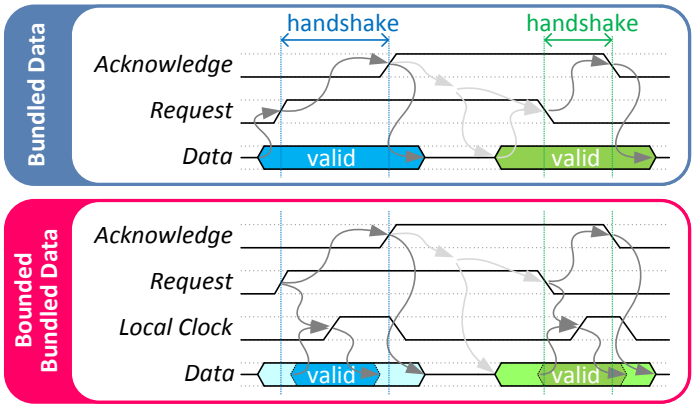


(a) Storage with datapath **(b)** Non-storage with datapath

Figure 6.1: Click storage and non-storage component implementations with bounded bundled data.



(a) A non-Storage component in-between two Storage components



(b) Timing diagram of protocols

Figure 6.2: The flipflop on the datapath receives a clocking signal from its local control module. Bundled Data protocol places valid data prior to the start of the handshake, and remains valid until the end of the handshake. Bounded Bundled Data protocol on the other hand allows the data to be skewed with respect to both request and acknowledge, as long as the data is valid for at least the setup time prior to the local clock's active edge and at least the hold time afterwards. From the timing diagram, the blue and green data shows the first and the second valid data.

Two-phase non-return-to-zero handshake protocol with *bundled data* is often used between consecutive components in a dataflow pipeline, where each module captures and stores data locally. The main motivation to store data locally is to

increase throughput. With local storage, a component can operate and forward its current data while its predecessor components already prepare and evaluate the next data. The disadvantage is that local data storage costs latency and dissipates energy.

The BBD protocol on the other hand accommodates dataflow pipelines where only a subset of the components capture and store data. It was proposed by Willem Mallon [21] and is used in the ARC's data-driven compiler, ARCwelder, and in the Click circuit family that he developed at Portland State University. His motivation for adding components that don't store data is to avoid the extra latency and energy for storage when throughput is limited by the combinational logic rather than by the control components. BBD protocol minimizes the cost for delay matching the bundled control and data by enabling *delay borrowing* between the handshake channels of non-Storage components.

6.2 Modeling Data

To model validity of data and distinguish whether the data is new, current, or stale, even when the data values don't change, we created two virtual signals used only for verification purposes. The two signals are called *bind* and *release*. We use an encoding such that "$bind \neq release$" represents *valid* data while "$bind = release$" represents *invalid* data.

Figure 6.3 shows an example of data validity with a timing diagram. The two signals *bind* and *release* start from the same state to each other. When new valid data is placed the *bind* bit flips. This can happen from currently valid or invalid data. The current data becomes old or stale upon arrival of the new valid data. When the current data goes stale, for instance because it is being reset by

a multiplexer or overwritten by new valid data, the *release* bit flips. The *release* signal can flip only when the current data goes from valid to invalid.

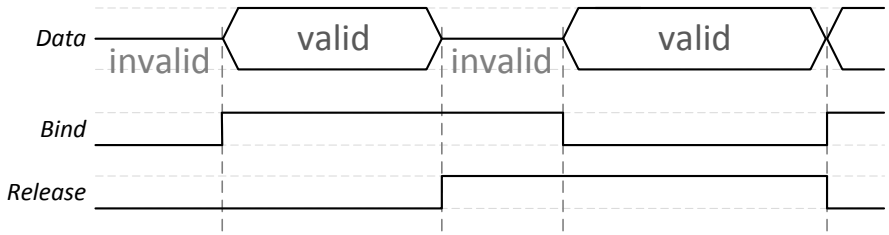


Figure 6.3: Data validity is modeled with *bind* and *release* signals. A flipping *bind* signals arrival of new valid data. A flipping *release* indicates that the current data is going stale. Data is valid if and only if $bind \neq release$.

Figure 6.4 shows that the flipflop on the datapath is expanded to accommodate *bind* and *release* signals. Since input D of the flipflop gets passed on to Q on each rising edge of the clock signal, the two virtual signals *bind* and *release* are flipped and passed on to the output to mark the data as a new valid data. We also add a placeholder for combinational logic shown as CL in front of the flipflop. Gate CL is similar to a cgate, but has additional *bind* and *release* signals along with variable *stop_release* which can be used in timing constraints to stop the output from releasing the data prematurely.

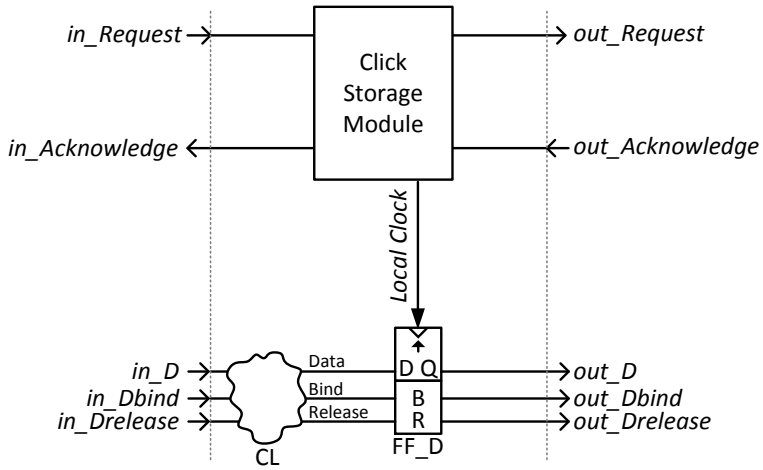


Figure 6.4: Click storage with datapath. The datapath has two virtual signals, *bind* and *release* to represent data validity. The data flipflop has been extended to pass on *bind* and *release* signals on a rising clock edge. Combinational logic (CL) in front of the data flipflop can have its own delay or even borrow delay from the predecessor component.

6.3 RT Constraints for BBD

The BBD protocol provides a large degree of independence for the design of the datapath into combinational logic and flipflops versus the design of the self-timed control. Data and control are mutually bounded only by the setup and hold times of the gates where they meet. Typical gates that act as points of convergence for data and control are locally clocked flipflops and multiplexers.

The setup constraint requires that data is ready before clocking the data flipflop. As shown in Figure 6.5, the predecessor's path delays are also considered and the predecessor's *and*+ becomes the POD. The EARLY event is where the new data is ready on the data flipflop's input D, which can be seen as $CL.Bind\pm$. The LATE event is where the data flipflop receives an active clock edge, buf_ck_D+.

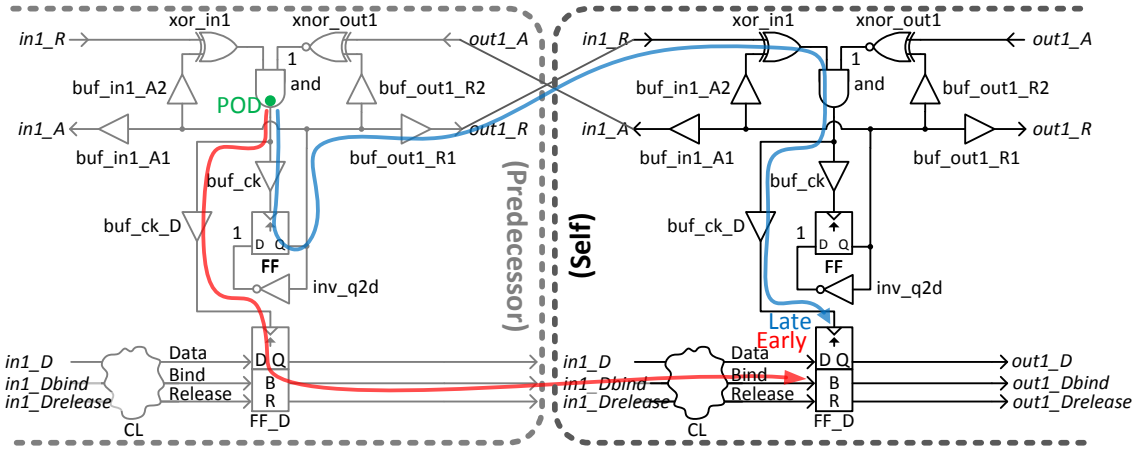


Figure 6.5: BBD flipflop setup constraint. The setup constraint simply states the data must be ready before clocking the flipflop. The POD for this constraint starts from the predecessor's *and* gate. The early path is colored in red and the late path in blue. BBD_setup: $Predecessor.and+ \rightarrow CL.Bind\pm \prec buf\_ck\_D+$

The hold constraint shown in Figure 6.6 requires that the time from $and+$ to clocking the data flipflop is shorter than the time going over the input channel to release and possibly replace the data. This can be characterized in terms of *bind* and *release* changes, requiring *release* to come only after the data has been clocked and stored into the data flipflop.

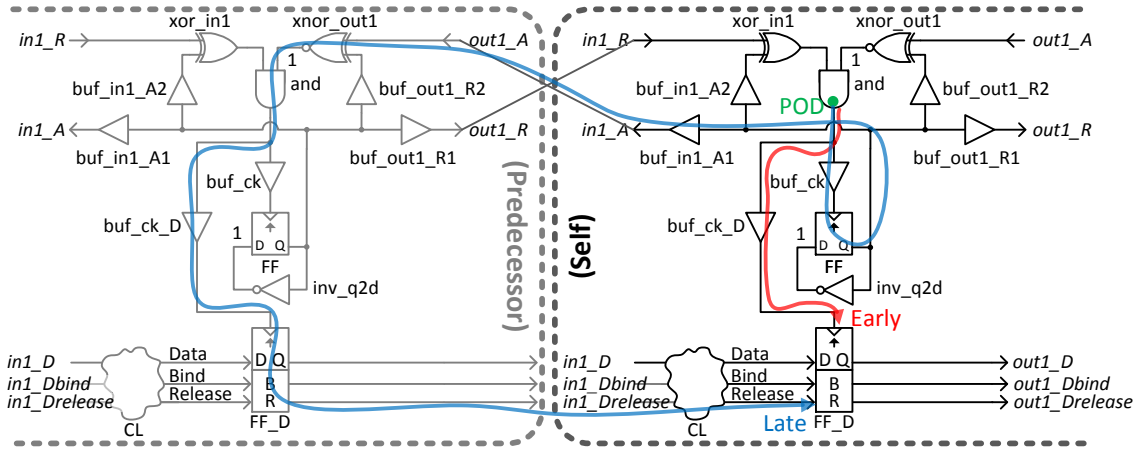


Figure 6.6: BBD flipflop hold constraint. The hold constraint states that the current data must have been picked up by clocking the data flipflop before releasing the data. BBD_hold: $and+ \rightarrow buf\_ck\_D+ \prec CL.Release\pm$

In summary, the additional RT constraints for the BBD datapath are as follows:

- (BBD setup) $Predecessor.and+ \rightarrow CL.Bind\pm \prec buf\_ck\_D+$
- (BBD hold) $and+ \rightarrow buf\_ck\_D+ \prec CL.Release\pm$

6.4 Code Changes and Additions in NuSMV

This section goes over the details and changes in the NuSMV code to accommodate BBD datapaths, and is based on the code from Chapter 5. With the validity of data being modeled using *bind* and *release* signals, we need to extend some of the NuSMV module definitions with *bind* and *release* information and add a few modules to track and verify *bind* and *release* changes. Key changes are as follows.

- Semimodularity check:

  The new NuSMV code distinguishes more gate varieties, each with its own semimodularity check. It makes sense to share the code for semimodularity checking by creating a parameterized module, *semimodular_check*, that can be instantiated as needed.

- BBD check:

  The changes in *bind* and *release* must match the data validity encoding illustrated in Figure 6.3. To verify that this is the case, we defined a new NuSMV module, *bbd_check*, which can be instantiated whenever a BBD check is required.

- Combinational gate and flipflop modules that use or produce BBD signals:

  These module variations must include *bind* and *release* signals.

- Circuit, Environment, and Main modules:

  The Circuit module contains the connectivity of the circuit while the Environment module models the channel communications with the circuit. The Main module pulls everything together. These modules must be adapted to accommodate and verify *bind* and *release* signals.

Figure 6.7 shows the code for detection of semimodularity violation. This function was part of the cgate module in Chapter 5 but we moved it out and created a stand-alone module, to enable code sharing.

The new module has four variables: *val, set, stop-rise,* and *stop-fall.* Variable *val* is the output and *set* is the value of the input function. To indicate whether the output transition is blocked by one or more RT constraints, parameters *stop-rise* and *stop-fall* are used to stop rising and falling transitions, respectively.

When the output is scheduled to make a transition and is not blocked by any RTs, but the input function changes and the output no longer needs to make a transition, semimodularity is flagged and *semimodular* becomes false. This is shown in line 8 in the case statement.

```
1  MODULE semimodular_check (val, set, stop_rise, stop_fall)
2     VAR
3        semimodular : boolean;
4     ASSIGN
5        init(semimodular) := TRUE;
6     TRANS
7        next(semimodular) = case
8           ((!stop_rise & !val & set) | (!stop_fall & val & !set)) & next(val=set) & next(val)=val : FALSE;
9           TRUE : semimodular;
10       esac;
11    --PROPERTIES
12    --safety
13    CTLSPEC AG semimodular
```

Figure 6.7: Semimodularity check. If the output transition was not blocked, and the output was about to make a transition but was canceled due to an input change, *semimodularity* becomes false. This module is used for all *cgates* as well as other types of gate modules in the flow control and datapath.

```
MODULE bbd_check (valid_set, set, D, bind, release)
   VAR
      bbd: boolean;
   ASSIGN
      init(bbd) := TRUE;
   DEFINE
      valid_D := (bind != release);
   TRANS
      next(bbd) = case
         --A change in "bind" indicates that
         --D is generated from a valid set input with a valid result value "set".
         !( bind != next(bind)                  -> (valid_set & next(D) = set) ) : FALSE;
         --A valid D remains stable until released
         !( (valid_D & release = next(release)) -> D = next(D) )                 : FALSE;
         --Bind changes exactly once from the current to the next [D-invalid to D-valid to D-invalid] cycle
         --(where D-invalid may coincide with the previous D-valid) - namely when D become valid.
         --Release changes also exactly once for each such cycle - namely when D become invalid.
         --Bind and release changing together indicates that the new valid data invalidate the old data.
         !( (valid_D & bind != next(bind))      -> release != next(release) )    : FALSE;
         !( !valid_D                            -> release  = next(release) )    : FALSE;
         TRUE: bbd;
      esac
   --PROPERTIES
   --safety
   CTLSPEC AG bbd
```

Figure 6.8: NuSMV code for BBD checking. This module is instantiated from each of modules in the datapath.

```
MODULE cgate_data (set, bind_set, release_set, init_val, stop_release)
--Inertial delay extension of the inertial-delay cgate, adding data and its bind and release info.
--The changes (X) on bind or release can be sensed and used in RT constraints for bundled-data.
--When blocked by an RT constraint, DATA MUST BE MAINTAINED.
--   The definition of semimodularity CHANGES again over the prior rt-aware control version!!!
--   * We allow changes in set for invalid data, i.e. set!=next(set) is OK, i.e. semimodular,
--     when bind_set=release_set (invalid-set-data).
--   * Semimodularity for bind and release remain as strict as before.
   VAR
      val         : boolean;
      bind        : boolean;
      release     : boolean;
   ASSIGN
      init(val)         := init_val;
      init(bind)        := FALSE;
      init(release)     := FALSE;
      next(val) := case
         stop_data : val;
         TRUE      : set;
      esac;
      next(bind) := case
         stop_data : bind;
         TRUE      : bind_set;
      esac;
      next(release) := case
         stop_data  : release;
         TRUE       : release_set;
      esac;
   DEFINE
      valid_set := bind_set != release_set;
      stop_data := stop_release & (release != release_set);
   VAR
   --PROPERTIES
   --safety
      --semimodularity
      --   semimodular_check (val, set, stop_rise & , stop_fall)
      --   NOTE: functions with invalid incoming data need not obey semimodularity
      semimodular_val     : semimodular_check (val    , set        , (stop_data | !valid_set), (stop_data | !
           valid_set));
      semimodular_bind    : semimodular_check (bind   , bind_set   , stop_data               , stop_data);
      semimodular_release : semimodular_check (release, release_set, stop_data               , stop_data);
      --BBD
      --   bbd_check (valid_set, set, D, bind, release)
      bbd_cgate : bbd_check (valid_set, set, val, bind, release);
   --progress
      FAIRNESS running
```

Figure 6.9: CL gate for generic combinational logic is an instantiation of cgate_data module. When this module process is selected, it passes on the *Data*, *Bind*, and *Release* as long as the transition is not blocked by one or more RT constraints.

Figure 6.8 shows the module for BBD check. This module is instantiated from each gate module in the datapath. It samples data, bind, and release from both the inputs and outputs and checks whether BBD is violated.

Figure 6.9 is the module that is instantiated as combinational logic (CL) in the datapath shown in Figure 6.4. This module is an extension of the inertial delay cgate module with support of data validity. Semimodularity checks are performed for handing over *Data*, *Bind*, and *Release* signals from CL input to CL output.

```
1  MODULE ff_posedge_data (ck1, d1, bind1, release1, init_q)
2  --Greedy posedge triggered flipflop for bounded bundled data, with bind and release bits.
3  --Flipflops release previous data while binding new data, so bind != release at all times.
4  --Without loss of generality (because it's the bind/release changes that count)
5  --we start with FF.bind TRUE and FF.release FALSE, for all FF.
6     VAR
7        --GATES
8        --   count_bind_release_pc (bind, release, guard)
9        count1 : count_bind_release_pc (bind1, release1, guard1);
10    VAR
11       q            : boolean;
12       bind         : boolean;
13       release      : boolean;
14    ASSIGN
15       init(q)             := init_q;
16       init(bind)          := TRUE;
17       init(release)       := FALSE;
18    TRANS
19       next(q) = case
20          guard1 : d1;
21          TRUE: q;
22       esac;
23    TRANS
24       next(bind) = case
25          guard1 : !bind;
26          TRUE: bind;
27       esac;
28    TRANS
29       next(release) = case
30          guard1 : !release;
31          TRUE: release;
32       esac;
33    DEFINE
34       guard1     := !ck1 & next(ck1);
35       valid_set1 := bind1 != release1 & count1.cnt_bind = 1 & (count1.cnt_release = 1 | count1.initial_cycle);
36    VAR
37    --PROPERTIES
38    --safety
39       --BBD
40       --   bbd_check (valid_set, set, D, bind, release)
41       bbd_ff : bbd_check (valid_set1, next(ck1) & d1, q, bind, release);
```

Figure 6.10: Flipflop model with BBD inputs. The flipflop is initialized with valid data. Upon each posedge clock, the *Bind* and *Release* flip, keeping the relation $Bind \neq Release$ at all times. The internal *count1* module verifies that no valid data coming in is skipped or stuttered as valid flipflop data going out on *ff_posedge_data.q*. It does this by counting the number of incoming *Bind* and *Release* changes per clock cycle. The code details for *count1* are available in Appendix C.

Figure 6.10 is a flipflop model for bundled data with additional *Bind* and *Release* bits. Flipflops release previous data while binding new data so it is always

the case that $bind \neq release$, which means that data stored and output by a flipflop is valid at all times.

Figures 6.11–6.13 shows the code for the circuit and RT constraints, the environment, and the main module that connects everything together. BBD setup constraint $rtbbd1$ and hold constraint $rtbbd2$ can be found in Figure 6.11.

- (BBD setup) $rtbbd1 : in1\_POD\pm \rightarrow CL.bind\pm \prec and\pm$
- (BBD hold) $rtbbd2 : and+ \rightarrow buf\_ck\_D\pm \prec CL.*\pm$

Notice that in Section 6.3 when there were two storage components, the POD event for BBD setup was *predecessor*'s AND gate, but now refers to $in1\_POD$ in the environment module. The LATE event buf_ck_D is now $and\pm$ which is earlier point in time on the LATE path for generalization. By the way, there are also some additional RTs we haven't discussed yet. This is because the control logic now forks the output of the AND gate to two clocks, FF and FF_D.

```
1  MODULE circuit (in1_R, in1_POD, in1_D, in1_Dbind, in1_Drelease, out1_A)
2     VAR
3        --CONTROL LOGIC
4        --Declaration format:
5        --   process cgate (set, init_val, lazy, stop_rise, stop_fall)
6        --   process cgate_data (set, bind_set, release_set, init_val, stop_release)
7        --   process ff_flip_on_posedge_w_guard_w_q2d (ck, guard, init_out)
8        --           ff_posedge_data (ck, d, dbind, drelease, init_q)
9        xor_in1    : process cgate (in1_R xor buf_in1_A2.val   , FALSE, FALSE, FALSE, FALSE);
10       xnor_out1  : process cgate (out1_A xnor buf_out1_R2.val, TRUE , FALSE, FALSE, FALSE);
11       and        : process cgate (xor_in1.val & xnor_out1.val, FALSE, FALSE, stop_and_HI, stop_and_LO);
12       buf_ck     : process cgate (and.val                    , FALSE, FALSE, FALSE, FALSE);
13       buf_ck_D   : process cgate (and.val                    , FALSE, FALSE, FALSE, FALSE);
14       FF         : process ff_flip_on_posedge_w_guard_w_q2d (buf_ck.val, TRUE, FALSE);
15       buf_in1_A1 : process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
16       buf_in1_A2 : process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
17       buf_out1_R1: process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
18       buf_out1_R2: process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
19       --DATAPATH LOGIC
20       CL         : process cgate_data (in1_D, in1_Dbind, in1_Drelease, FALSE, stop_CLrelease_x);
21       FF_D       : ff_posedge_data (buf_ck_D.val, CL.val, CL.bind, CL.release, FALSE);
22    DEFINE
23       in1_A         := buf_in1_A1.val;
24       out1_R        := buf_out1_R1.val;
25       out1_D        := FF_D.q;
26       out1_Dbind    := FF_D.bind;
27       out1_Drelease := FF_D.release;
28    VAR
29       --CONSTRAINTS
30       --Declaration instance:
31       --  rt (eventPOD, eventEARLY, init_rt, guardPOD, guardEARLY, guardLATE, xPOD, xEARLY)
32       --(1) clock domain FF: and/ clocks the control FF.
33       --      POD   : and/
34       --      Early : {xor_in_n1\, xnor_out_m1\, FF.d X, buf_ck\, buf_ck_D\}
35       --      Late  : {in_n2_R X, out_m2_A X}
36       --      Repair: at each failing late event
37       rtc1: rt (and.val, !xor_in1.val  , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
38       rtc2: rt (and.val, !xnor_out1.val, GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
39       rtc3: rt (and.val, FF.d          , GREEN, TRUE, TRUE, TRUE, FALSE, TRUE);
40       rtc4: rt (and.val, !buf_ck.val   , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
41       rtc5: rt (and.val, !buf_ck_D.val , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
42       --(2) Bounded Bundled Data Setup
43       --    POD   : in_n1_POD X
44       --            NOTE: for its out-channels, which are in-channels to SUCC modules, POD=and/
45       --    Early : {CL.bind X}
46       --    Late  : {and/}
47       --            NOTE: is replaceable by {and X} for easy STA translation
48       --    Repair: at in_n1_R
49       rtbbd1: rt (in1_POD, CL.bind, GREEN, TRUE, TRUE, TRUE, TRUE, TRUE);
50       --(3) Bounded Bundled Data Release
51       --    Note:
52       --       For proper release of data, we require that:
53       --       The time from and/ is shorter to FFD.ck/ than over the channel to a new in1_D value.
54       --       Thanks to rtd1, we already know that the first in1_D value came before and/.
55       --       and by assumption-commitment reasoning (OK for pred then OK for succ)
56       --       the release of it comes after and/ and before the next binding of in1_D.
57       --       The corresponding constraint parameters are:
58       --    POD   : and/
59       --    Early : {buf_ck_D/}
60       --            NOTE: by rtc5, this is replaceable by {buf_ck_D X} for easy STA translation
61       --    Late  : {CL.release X}
62       --    Repair: at late event
63       rtbbd2: rt (and.val, buf_ck_D.val, GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
64       --(4) Left-over isochronic forks, pushed to the point of semimodularity:
65       --    POD   : and/
66       --    Early : {buf_ck_D/}
67       --            NOTE: by rtc5, this can be replaced by {buf_ck_D X} for easy STA translation,
68       --    Late  : {and\}
69       --            NOTE: by rtc5, this can be replaced by {and X} for easy STA translation
70       --    Repair: at buf_in1_A2/buf_out1_R2
71       --            NOTE: this internal repair is likely never needed.
72       rtf1: rt (and.val, buf_ck_D.val, GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
73    DEFINE
74       --Combine RT's with the same late events into one RT with the name of the late events
75       stop_c          := rtc1.stop | rtc2.stop | rtc3.stop | rtc4.stop | rtc5.stop;
76       stop_in1R_x     := stop_c;
77       stop_out1A_x    := stop_c;
78       stop_CLrelease_x:= rtbbd2.stop;
79       stop_and_HI     := rtbbd1.stop;
80       stop_and_LO     := rtf1.stop;
81    --PROPERTIES
82    --progress
83    FAIRNESS running
```

Figure 6.11: Module *circuit* has all the connectivity information for the control logic. All RT constraints' POD and EARLY, and some LATE points are in this module. BBD setup constraint is shown *rtbbd1* in line 49, where POD is $in1\_POD$, EARLY is $CL.bind$. The LATE event is $and\pm$ seen in line 11.

```
MODULE ENVin_data (in_A, random_inD, stop_inR_x, stop_inDrelease_x)
   VAR
      --GATES
      --   cgate        (set, init_val, lazy, stop_rise, stop_fall)
      --   ff_doubledge (ck, d, init_q)
      buf_inA    : process cgate (in_A         , FALSE, TRUE , stop_inDrelease_x, stop_inDrelease_x);
      inv_POD    : process cgate (!buf_inA.val , FALSE, TRUE , FALSE            , FALSE);
      buf_inR    : process cgate (inv_POD.val  , FALSE, FALSE, stop_inR_x       , stop_inR_x);
      FF_inD     : ff_doubledge  (inv_POD.val  , random_inD  , FALSE);
   DEFINE
      in_POD     := inv_POD.val;
      in_R       := buf_inR.val;
      in_D       := FF_inD.q;
      in_Dbind   := inv_POD.val;
      in_Drelease:= buf_inA.val;
   VAR
   --PROPERTIES
   --safety
      --BBD
      --   bbd_check (valid_set, set, D, bind, release)
      bbd_ENVin : bbd_check (TRUE, random_inD, in_D, in_Dbind, in_Drelease);
   --progress
      FAIRNESS running
```

(a) Module *ENVin_ data* is a submodule for the environment on the input channel.

```
MODULE environment (in1_A, out1_R, out1_D, out1_Dbind, out1_Drelease, stop_in1R_x, stop_in1Drelease_x, stop_out1A_x)
   VAR
      --GATES:
      --Reminder of module definitions:
      --  ENVin_data (in_A, random_inD, stop_inR_x, stop_inDrelease_x)
      --  cgate (set, init_val, lazy, stop_rise, stop_fall)
      ENV_in1       : process ENVin_data (in1_A, random_in1D, stop_in1R_x, stop_in1Drelease_x);
      ENV_out1      : process cgate (out1_R, FALSE, TRUE, stop_out1A_x, stop_out1A_x);
      random_in1D   : boolean;
   DEFINE
      in1_R         := ENV_in1.in_R;
      in1_POD       := ENV_in1.in_POD;
      in1_D         := ENV_in1.in_D;
      in1_Dbind     := ENV_in1.in_Dbind;
      in1_Drelease  := ENV_in1.in_Drelease;
      out1_A        := ENV_out1.val;
   --PROPERTIES
   --progress
   FAIRNESS running
```

(b) Module *environment* interacts with the circuit module. Environment is part of a LATE event for one or more RT constraints. Random inputs are supplied for the data.

Figure 6.12: Environment produces inputs and accepts outputs from the circuit.

```
MODULE main
   VAR
      StorageProtocol   : protocol (in1_R, in1_A, out1_R, out1_A);
      StorageEnvironment: process environment (in1_A, out1_R, out1_D, out1_Dbind, out1_Drelease, stop_in1R_x,
           stop_in1Drelease_x, stop_out1A_x);
      StorageCircuit    : process circuit (in1_R, in1_POD, in1_D, in1_Dbind, in1_Drelease, out1_A);
   DEFINE
      in1_R              := StorageEnvironment.in1_R;
      in1_A              := StorageCircuit.in1_A;
      in1_POD            := StorageEnvironment.in1_POD;
      in1_D              := StorageEnvironment.in1_D;
      in1_Dbind          := StorageEnvironment.in1_Dbind;
      in1_Drelease       := StorageEnvironment.in1_Drelease;
      out1_R             := StorageCircuit.out1_R;
      out1_A             := StorageEnvironment.out1_A;
      out1_D             := StorageCircuit.out1_D;
      out1_Dbind         := StorageCircuit.out1_Dbind;
      out1_Drelease      := StorageCircuit.out1_Drelease;
      stop_in1R_x        := StorageCircuit.stop_in1R_x;
      stop_in1Drelease_x := FALSE;
      stop_out1A_x       := StorageCircuit.stop_out1A_x;
   --PROPERTIES
   --progress
   FAIRNESS running
```

Figure 6.13: Module *main* connects the *protocol*, the *environment*, and the *circuit*.

6.5 Counter Examples without BBD Constraints

The final schematic of the Click storage with datapath is shown in Figure 6.14.

Figure 6.15 shows two counterexamples for the gate-level circuit diagram in Figure 6.14 when data setup and hold are not constrained.

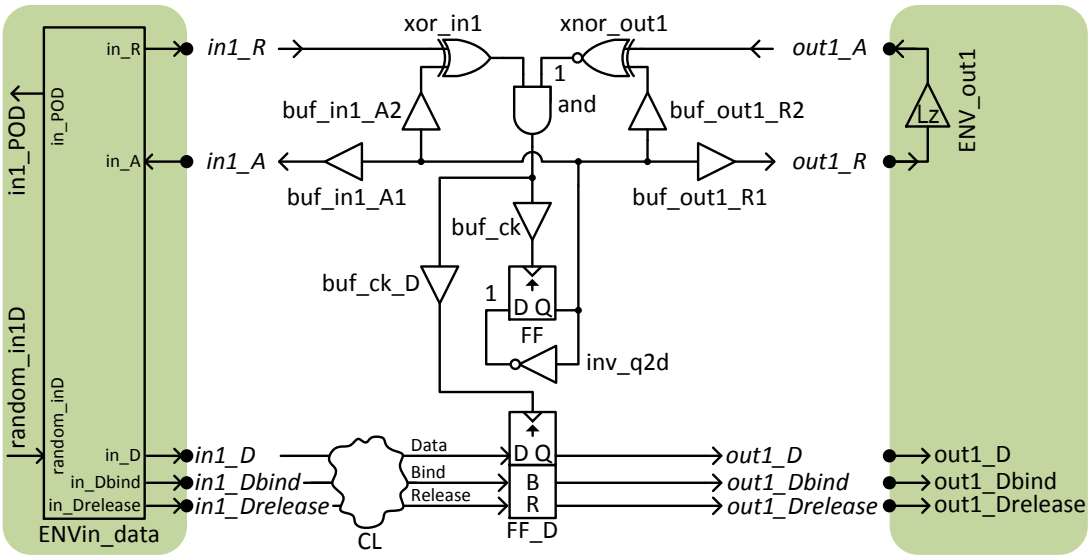


Figure 6.14: Circuit and Environment model of Click Storage with datapath. Corresponding module definitions are coded in Figures 6.11–6.13.

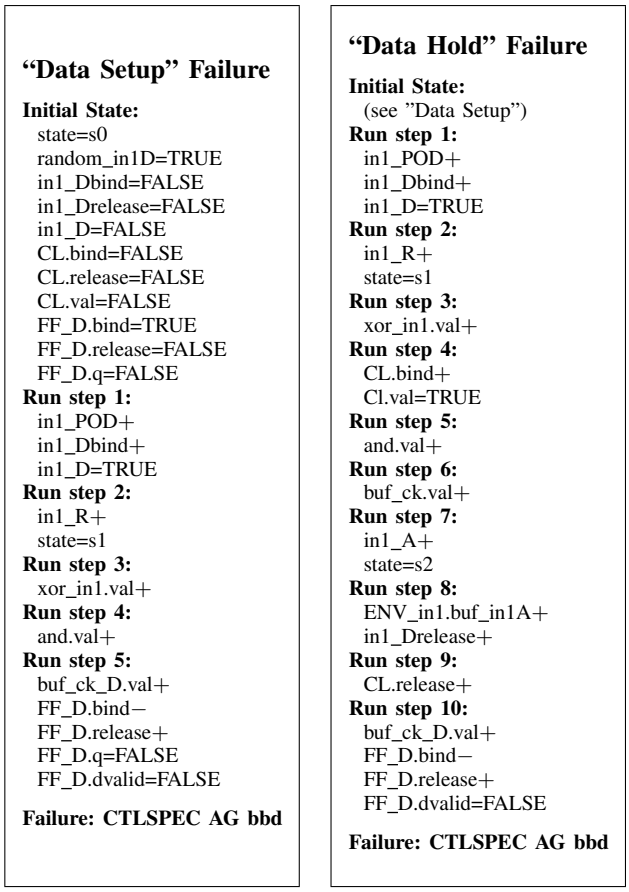


Figure 6.15: Click Storage circuit with datapath and environment coded in Figures 6.11–6.12 and two counterexamples created by NuSMV. Corresponding gate-level circuit diagram is in Figure 6.14. As before, in Figure 5.18, the counterexamples each describe a path of events through a not-yet correctly constrained circuit. The left counterexample shows a "data setup" failure at data flipflop *FF_D*, created in absence of relative timing constraint *rtbbd1*. In the counterexample, the data coming in through channel *in1* lag behind too much and fail to make it through the combinational logic, *CL*, before the logic results are captured into *FF_D* at the rising clock edge, *buf_ck_D+*. As a result, flipflop *FF_D* captures invalid data, which is detected by the *bbd_check* that we built into the NuSMV code of the flipflop. The right counterexample shows a "data hold" failure at data flipflop *FF_D*, created in absence of relative timing constraint *rtbbd2*. In this case, the data coming in through channel *in1* do make it through the combinational logic, *CL*, but are released too soon. The incoming data are released and the released data are propagated through the logic before the logic results are captured into *FF_D*. Consequently *FF_D* may have invalid data, which is detected by the *bbd_check* in the NuSMV code of the flipflop.

6.6 Non-Storage Component

As explained earlier in Section 6.1, non-storage components are used where it is known that the previous component is holding data. The only difference from the storage component shown in Section 6.4 is that it lacks a local data flipflop along with the associated buffer for the clock. Because there is no data flipflop, the extra fork constraint "*rtf1*" in Figure 6.11 related to the data flipflop clock is no longer needed. Other than that, the RT constraints for control remains unchanged.

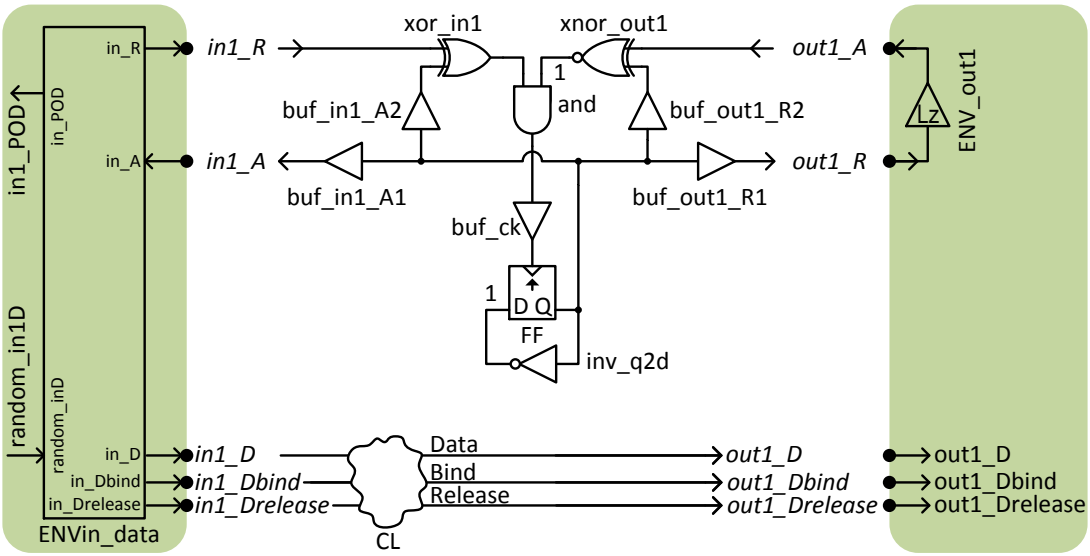


Figure 6.16: Circuit and Environment model of Click non-Storage with datapath. The datapath only has a CL module and no data flipflop.

Figure 6.16 shows the circuit and environment model of the non-storage datapath model, and Figure 6.17 shows the corresponding NuSMV code for the *circuit* portion of the component. The BBD constraints are also reduced to just one constraint, *rtbbd* seen in Figure 6.17 line 49.

- (BBD) $rtbbd1 : in1\_POD\pm \rightarrow CL.bind\pm \prec in1\_Drelease\pm$

This is a much weaker constraint than the *rtbbd1* and *rtbbd2* from the previous storage version. When this non-storage component is sitting between two storage componets, the *rtbbd1* and *rtbbd2* of the storage component that captures the output of the non-storage component basically already covers this weak constraint. So in most cases, this constraint will not be used.

The environment now gets a constraint because the semimodularity for the *release* is pushed back to the environment and now it becomes a *STOP* signal as a LATE event.

```
1  MODULE circuit (in1_R, in1_POD, in1_D, in1_Dbind, in1_Drelease, out1_A)
2     VAR
3        --CONTROL LOGIC
4        --Executed as part of the (FAIR) process interleaving schedule
5        --Declaration format:
6        --   process cgate (set, init_val, lazy, stop_rise, stop_fall)
7        --   process cgate_data (set, bind_set, release_set, init_val, stop_release)
8        --   process ff_flip_on_posedge_w_guard_w_q2d (ck, guard, init_out)
9        xor_in1    : process cgate (in1_R xor buf_in1_A2.val   , FALSE, FALSE, FALSE, FALSE);
10       xnor_out1  : process cgate (out1_A xnor buf_out1_R2.val, TRUE , FALSE, FALSE, FALSE);
11       and        : process cgate (xor_in1.val & xnor_out1.val, FALSE, FALSE, FALSE, FALSE);
12       buf_ck     : process cgate (and.val                    , FALSE, FALSE, FALSE, FALSE);
13       FF         : process ff_flip_on_posedge_w_guard_w_q2d (buf_ck.val, TRUE, FALSE);
14       buf_in1_A1 : process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
15       buf_in1_A2 : process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
16       buf_out1_R1: process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
17       buf_out1_R2: process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
18       --DATAPATH LOGIC
19       CL         : process cgate_data (in1_D, in1_Dbind, in1_Drelease, FALSE, FALSE);
20    DEFINE
21       in1_A         := buf_in1_A1.val;
22       out1_R        := buf_out1_R1.val;
23       out1_D        := CL.val;
24       out1_Dbind    := CL.bind;
25       out1_Drelease := CL.release;
26    VAR
27       --CONSTRAINTS
28       --Declaration instance:
29       --  rt (eventPOD, eventEARLY, init_rt, guardPOD, guardEARLY, guardLATE, xPOD, xEARLY)
30       --(1) clock domain FF: and/ clocks the control FF.
31       --      POD   : and/
32       --      Early : {xor_in_n1\, xnor_out_m1\, FF.d X, buf_ck\}
33       --      Late  : {in_n2_R X, out_m2_A X}
34       --      Repair: at each failing late event
35       rtc1: rt (and.val, !xor_in1.val  , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
36       rtc2: rt (and.val, !xnor_out1.val, GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
37       rtc3: rt (and.val, FF.d          , GREEN, TRUE, TRUE, TRUE, FALSE, TRUE);
38       rtc4: rt (and.val, !buf_ck.val   , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
39       --(2) Bounded Bundled Data Setup and Release
40       --    ini_POD X -> CL.bind X < ini_Drelease X
41       --    INTERPRETATION:
42       --       Before releasing the data on channel ini,
43       --       ensure that these data have produced a valid combinational logic result"
44       --    Default repair : at late events
45       --    STA translation:
46       --       POD: ini_POD X
47       --       -> early: {CL.bind X} ~ maxpath{POD to in_D X to CL.val X}
48       --       -> late: {ini.Drelease X} ~ minpath{POD to in_R X to in_A X to in_D X}
49       rtbbd: rt (in1_POD, CL.bind, GREEN, TRUE, TRUE, TRUE, TRUE, TRUE);
50    DEFINE
51       --Combine RT's with the same late events into one RT with the name of the late events
52       stop_c             := rtc1.stop | rtc2.stop | rtc3.stop | rtc4.stop;
53       stop_in1R_x        := stop_c;
54       stop_out1A_x       := stop_c;
55       stop_in1Drelease_x := rtbbd.stop;
56    --PROPERTIES
57    --progress
58    FAIRNESS running
```

Figure 6.17: Module *circuit* for non-storage. There's no data flipflop in the datapath.

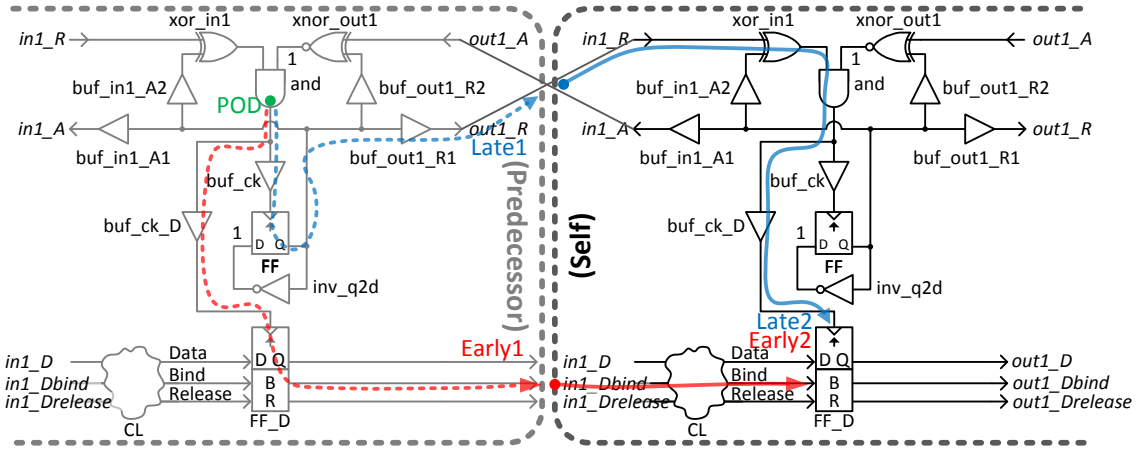
```
1  MODULE main
2     VAR
3        StorageProtocol   : protocol (in1_R, in1_A, out1_R, out1_A);
4        StorageEnvironment: process environment (in1_A, out1_R, out1_D, out1_Dbind, out1_Drelease, stop_in1R_x,
               stop_in1Drelease_x, stop_out1A_x);
5        StorageCircuit    : process circuit (in1_R, in1_POD, in1_D, in1_Dbind, in1_Drelease, out1_A);
6     DEFINE
7        in1_R              := StorageEnvironment.in1_R;
8        in1_A              := StorageCircuit.in1_A;
9        in1_POD            := StorageEnvironment.in1_POD;
10       in1_D              := StorageEnvironment.in1_D;
11       in1_Dbind          := StorageEnvironment.in1_Dbind;
12       in1_Drelease       := StorageEnvironment.in1_Drelease;
13       out1_R             := StorageCircuit.out1_R;
14       out1_A             := StorageEnvironment.out1_A;
15       out1_D             := StorageCircuit.out1_D;
16       out1_Dbind         := StorageCircuit.out1_Dbind;
17       out1_Drelease      := StorageCircuit.out1_Drelease;
18       stop_in1R_x        := StorageCircuit.stop_in1R_x;
19       stop_in1Drelease_x := StorageCircuit.stop_in1Drelease_x;
20       stop_out1A_x       := StorageCircuit.stop_out1A_x;
21    --PROPERTIES
22    --progress
23    FAIRNESS running
```

Figure 6.18: Module *main* for non-storage. The only difference is that there is *stop_in1_Drelease_x* = *StorageCircuit.stop_in1_Drelease_x* which is used to stop the environment as part of the LATE constraint.

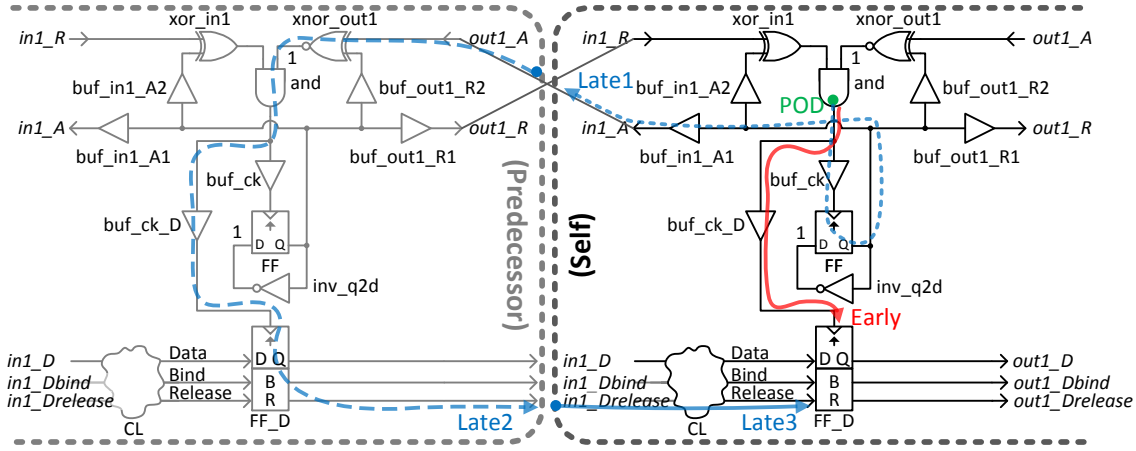
6.7 STA Translation

As explained in Section 5.2.5, ARCtimer does static timing analysis (STA) in a modular fashion through procedure calls and calculate the delays of paths from the RT constraints while being aware of going through flipflops and channels, which are marked as checkpoints. The checkpoint-to-checkpoint pieces of these paths are stored in the component design library, on a per-component basis.

Figure 6.19 is a repeat of Figure 6.5 and 6.6, but shows how the RT paths are broken into pieces for STA. Typically, maximum delay is calculated for the EARLY path, while minimum delay is calculated for the LATE path. As long as the maximum delay of the early path is less than the minimum delay of the late path, there is nothing to fix. ARCtimer knows which procedure to instantiate. Though it may look like the pats are cut at checkpoints, ARCtimer can, in principle, glue the various component pieces back together seamlessly into the final paths for STA.



(a) BBD flipflop setup constraint for STA. The two paths are cut into pieces when going over a channel.



(b) BBD flipflop hold constraint for STA. The two paths are cut into pieces when going over a channel.

Figure 6.19: BBD constraints from Figure 6.5 and 6.6, but with the paths cut at the channels for STA.

7

Conclusion

With the advancement of technology, digital circuits are becoming more complex and it is becoming harder to manage the whole system with a global clock. Asynchronous circuits operating on handshake protocols instead of a global clock has the potential to bring better power efficiency, high speed, and robustness. However, the lack of tool and support from the industry for asynchronous design has been and still is a major hurdle for wide adoption.

Timing verification in asynchronous system is especially important since handshake communications are driven by signal transitions on the wires rather than being sampled at fixed intervals of an active clock edge. Since there is no synchronization of time, hazards must be avoided.

Delay insensitive circuits, which are the most robust type of asynchronous circuits, must operate correctly with unknown delay in wires and gates. This type of circuit are hazard-free, but due to heavy restriction, only a small number of circuits are truly insensitive to delays.

This thesis tackles timing closure of asynchronous circuits and presents a framework for generating and verifying timing constraints for handshake components that use bounded-bundled-data handshake protocols. We add timing constraints where needed to make the circuit follow a delay-insensitive specification. The timing constraints that we add are based on relative timing methodology, where select events are ordered relative to another event. We have expanded the capability of relative timing by including guard conditions that can be used to

specify a sequence of events. The constraints are not tied with physical process technology so the constraints can be carried over. To verify that the timing constraint set is complete, we used a general purpose model checker, and show how to model the system, the timing constraints, the delay-insensitive specification, data validity, and how to check the properties that the circuit must satisfy. Because components in a same circuit family shares similarities, the complete set of timing constraints can be generalized into a pattern constraint and stored in the component library where designers can use the component without re-doing timing verification at the component level. For static timing analysis, the pattern constraints are broken down into STA tool friendly format. Essential decision points and choices one can make compared to others have been identified throughout the context.

I have applied ARCtimer to generate timing constraints for Click and GasP self-timed circuit families, using ARCtimer as indicated in Chapters 5–6. Appendix A in this thesis shows the modeling details and the generated timing constraints for a representative set of Click components. The Click component set is representative in that it contains components with parallel and with sequential handshake behaviors, with and without data storage, and with data-driven deterministic as well as arbitrated non-deterministic dataflow control. Appendix B presents tables that quantify the space and time complexity of verifying this set of Click components. Appendix C shows the NuSMV library modules that were used in generating and verifying the timing constraints of these Click components. Appendix D repeats the NuSMV code for the key example in Chapters 5–6: the Click Storage component, with datapath.

For Future work, improving our current automation on translating relative timing constraints to STA code would be helpful to speed up the completion of the design library. Increasing modeling capacity by adding theorem proving such as ACL2 would also be an interesting topic. Another future work could be applying the solutions presented in this thesis to so-called *Naturalized Communication* [39] versions of various circuit families. Naturalized communication separates components into links and joints and a standard link-join interface.

References

[1] Khaled Alsayeg, Katell Morin-Allory, and Laurent Fesquet. RAT-Based Formal Verification of QDI Asynchronous Controllers. In *Forum on Specification and Design Languages (FDL)*, pages 1–6, 2009.

[2] Peter Beerel, Georgios Dimou, and Andrew Lines. Proteus: An ASIC Flow for GHz Asynchronous Designs. *IEEE Design & Test of Computers*, 28(5):36–51, 2011.

[3] Peter Beerel, Recep Ozdag, and Marcos Ferretti. *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, NY, USA, 2010.

[4] Peter Beerel and Marly Roncken. Low Power and Energy Efficient Asynchronous Design. *Journal of Low Power Electronics (JOLPE)*, 3(3):234–253, 2007.

[5] Kees van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken, Frits Schalij, and Ad Peeters. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design & Test of Computers*, 11(2):22–32, 1994.

[6] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. NuSMV 2.4 User Manual, 2013. Download from http://nusmv.fbk.eu/NuSMV/userman/index-v2.html.

[7] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 2000.

[8] Jordi Cortadella, Mike Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, Berlin, Heidelberg, NewYork, 2002.

[9] Krishnaji Desai, Kenneth Stevens, and John O'Leary. Symbolic Verification of Timed Asynchronous Hardware Protocols. In *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 147–152, Aug 2013.

[10] Daniel Dobberpuhl, Richard Witek, Randy Allmon, Robert Anglin, Sharon Britton, Linda Chao, Robert Conrad, Daniel Dever, Bruce Gieseke, Gregory Hoeppner, John Kowaleski, Kathryn Kuchler, Maureen Ladd, Michael Leary, Liam Madden, Edward Mclellan, Derrick Meyer, James Montanaro, Donald Priore, Vidya Rajagopalan, Sridhar Samudrala, and Sribalan Santhanam. A 200 mhz 64 b dual-issue cmos microprocessor. In *Solid-State Circuits Conference, 1992. Digest of Technical Papers. 39th ISSCC, 1992 IEEE International*, pages 106–107, Feb 1992.

[11] Doug Edwards and Andrew Bardsley. Balsa: An Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, 2002.

[12] Robert Fuhrer and Steven Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. Kluwer Academic Publishers, Boston, MA, USA, 2001.

[13] David Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, 1954.

[14] Mark Josephs and Jan Tijmen Udding. An Algebra for Delay-Insensitive Circuits. In *Computer Aided Verification (CAV)*, pages 343–352, 1991.

[15] Prasad Joshi. Static Timing Analysis of GasP. Master's thesis, Electrical Engineering, University of Southern California, USA, December 2008.

[16] Sean Keller, Michael Katelman, and Alain Martin. A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits. In *Proc. Asynchronous Circuits and Systems (ASYNC)*, pages 65–76, 2009.

[17] Joep Kessels and Paul Marston. Designing Asynchronous Standby Circuits for a Low-Power Pager. *Proceedings of the IEEE*, 87(2):257–267, 1999.

[18] Hoshik Kim, Peter Beerel, and Ken Stevens. Relative Timing Based Verification of Timed Circuits and Systems. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 115–124, 2002.

[19] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[20] Willem Mallon. Theories and Tools for the Design of Delay-Insensitive Communicating Processes. PhD thesis, University of Groningen, The Netherlands, 2000.

[21] Willem Mallon. Bounded Bundled Data. Internal Report, ARC2012-hp02, Asynchronous Research Center (ARC), Portland State University, 2011. Available from our web site at http://arc.cecs.pdx.edu/publications.

[22] Alain Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In *Proc. Advanced Research in VLSI*, pages 263–278, 1990.

[23] Alain Martin and Mika Nyström. Asynchronous Techniques for System-on-Chip Design. *Proceedings of the IEEE*, 94(6):1089–1120, 2006.

[24] Kenneth McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[25] Teresa Meng. *Synchronization Design for Digital Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[26] Swetha Mettala Gilla. Library Characterization and Static Timing Analysis of Single-Track Circuits in GasP. Master's thesis, Electrical and Computer Engineering, Portland State University, USA, 2010.

[27] Raymond Miller. *Switching Theory Volume 2: Sequential Circuits and Machines, Chapters 9–10*. John Wiley & Sons, New York, USA, 1965.

[28] Dharmendra Modha. Introducing a brain-inspired computer, 2015. http://www.research.ibm.com/articles/brain-chip.shtml.

[29] David E. Muller and W. Scott Bartky. *A Theory of Asynchronous Circuits*. Harvard University Press, Cambridge, MA, USA, 1959.

[30] Radu Negulescu. Process Spaces and Formal Verification of Asynchronous Circuits. PhD thesis, University of Waterloo, Canada, 1998.

[31] Radu Negulescu and Ad Peeters. Verification of Speed-Dependences in Single-Rail Handshake Circuits. In *Proc. Asynchronous Circuits and Systems (ASYNC)*, pages 159–170, 1998.

[32] Hoon Park, Anping He, Marly Roncken, and Xiaoyu Song. Semi-modular delay model revisited in context of relative timing. *Electronics Letters*, 51(4):332–334, 2015.

[33] Hoon Park, Anping He, Marly Roncken, Xiaoyu Song, and Ivan Sutherland. Modular timing constraints for delay-insensitive systems. *Journal of Computer Science and Technology, Springer*, Accepted for publication, 2016.

[34] Marco Peña, Jordi Cortadella, Alex Kondratyev, and Enric Pastor. Formal verification of safety properties in timed circuits. In *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 2–11, 2000.

[35] Ad Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Proefschrift Technische Universiteit Eindhoven, Eindhoven, Netherlands, 1996.

[36] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click Elements: An Implementation Style for Data-Driven Compilation. In *Proc. Asynchronous Circuits and Systems (ASYNC)*, pages 3–14, 2010.

[37] Mallika Prakash. Library characterization and static timing analysis of asynchronous circuits. Master's thesis, University of Southern California, CA, USA, December 2007.

[38] Mallika Prakash and Peter Beerel. Static Timing Analysis of Template-Based Asynchronous Circuits. US Patent US 2009/0210841 A1, assigned to the University of Southern California, August 2009.

[39] Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland. Naturalized communication and testing. In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pages 77–84, May 2015.

[40] Sachin Sapatnekar. Chapter 6: Static Timing Analysis. In *L. Scheffer, L. Lavagno, G. Martin (Eds.): Electronic Design Automation for Integrated Circuits Handbook, Volume 2*. CRC Press, 2006.

[41] Louis Scheffer, Luciano Lavagno, and Grant Martin (Eds.). *Electronic Design Automation for Integrated Circuits Handbook, Volumes 1–2*. CRC Press and Taylor & Francis, 2006.

[42] Basit Riaz Sheikh and Rajit Manohar. An Operand-Optimized Asynchronous IEEE 754 Double-Precision FLoating-Point Adder. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 151–162, 2010.

[43] Basit Riaz Sheikh and Rajit Manohar. Energy-Efficient Pipeline Templates for High-Performance Asynchronous Circuits. *ACM Journal on Emerging Technologies in Computing Systems*, 7(4), 2011.

[44] Basit Riaz Sheikh and Rajit Manohar. An Asynchronous FLoating-Point Multiplier. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 89–96, 2012.

[45] Montek Singh and Steven M. Nowick. MOUSETRAP: High-speed Transition-Signaling Asynchronous Pipelines. *IEEE Transactions on Very Large Integration (VLSI) Systems*, 15(6):684–698, 2007.

[46] Jens Sparsø and Steve Furber (Eds.). *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, Boston, MA, USA, 2001.

[47] Ken Stevens and Yang Xu. Analyze and Artist: Tool Suite for Generating Relative Timing Constraints for Self-Timed Circuits Using a Bisimulation Equivalence Model, University of Utah.

[48] Kenneth Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. PhD thesis, University of Calgary, September 1994.

[49] Kenneth Stevens, Ran Ginosar, and Shai Rotem. Relative timing. In *Proc. Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, 1999.

[50] Kenneth Stevens, Ran Ginosar, and Shai Rotem. Relative Timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):129–140, 2003.

[51] Kenneth Stevens, Shai Rotem, Ran Ginosar, Peter Beerel, Chris Myers, Kenneth Yun, Rakefet Kol, Charles Dike, and Marly Roncken. An Asynchronous Instruction Length Decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, 2001.

[52] Kenneth Stevens, Yang Xu, and Vikas Vij. Characterization of Asynchronous Templates for Integration into Clocked CAD Flows. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 151–161, 2009.

[53] Ivan Sutherland. GasP Circuits that Work. ECE 507 Research Seminar, Fall 2010. Asynchronous Research Center, Portland State University. Download from http://arc.cecs.pdx.edu/fall10.

[54] Ivan Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[55] Ivan Sutherland and Scott Fairbanks. GasP: A Minimal FIFO Control. In *Proc. Asynchronous Circuits and Systems (ASYNC)*, pages 46–53, 2001.

[56] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1999.

[57] Vivek Tiwari, Deo Singh, Suresh Rajgopal, Gaurav Mehta, Rakesh Patel, and Franklin Baez. Reducing power in high-performance microprocessors. In *Design Automation Conference, 1998. Proceedings*, pages 732–737, June 1998.

[58] Jeanne Trinko. IBM, Keynote Speech, ASYNC 2013.

[59] Tom Verhoeff. A Theory of Delay-Insensitive Systems. PhD thesis, Eindhoven University of Technology, The Netherlands, 1994.

[60] Victor Varshavsky (Ed.). *Self-Timed Control of Concurrent Processes*. Kluwer Academic Publishers, Norwell, MA, USA, 1990.

[61] Vikas Vij. *Algorithms and Methodology to Design Asynchronous Circuits Using Synchronous CAD Tools and Flows*. PhD thesis, Electrical and Computer Engineering, The University of Utah, USA, 2013.

[62] Yang Xu. Algorithms for Automatic Generation of Relative Timing Constraints. PhD thesis, The University of Utah, USA, 2011.

[63] Yang Xu and Kenneth Stevens. Automatic Synthesis of Computation Interference Constraints for Relative Timing Verification. In *IEEE International Conference on Computer Design (ICCD)*, pages 16–22, 2009.

[64] Tomohiro Yoneda, Tomoya Kitai, and Chris Myers. Automatic Derivation of Timing Constraints by Failure Analysis. In *Proc. Computer Aided Verification (CAV)*, pages 195–208, 2002.

Appendix A

Click Family - Protocols, Circuits, Timing Patterns

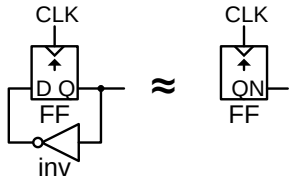
At the Asynchronous Research Center, we use the following three variations of a component:

- Parallel storage: Used as beginning and end components in a dataflow pipeline, to store data locally and maintain this data for entire duration of the dataflow operation. This component variation fills output channels and drain input channels in parallel.

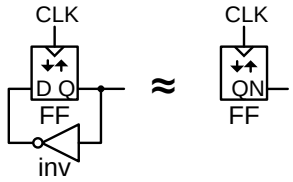
- Parallel non-storage: Used between parallel and storage components, where the combinational logic in the datapath may have more delay than the control path. Non-storage components do not store data locally, thus saving area, time, and power.

- Telescope non-storage: Used between parallel storage components. Telescope components do store data locally, to save area and power. This component variation first fills output channel and it drains input channels only after the output channels have been filled as well as drained.

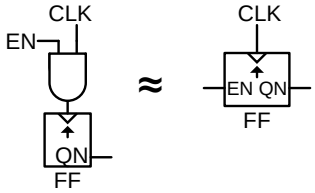
In the Click circuit family, it is common to tie the Q output of a flipflop via an inverter to the D input. Figure A.1 shows a *positive-edge flipping flipflop*. Triggering of a flipflop can be made conditionally by adding an EN signal that gates either the Q to D propagation or the CLK propagation.



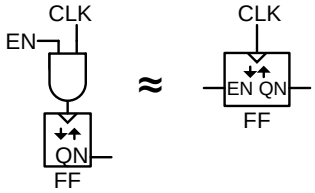
(a) Positive-edge flipping flipflop



(b) Double-edge flipping flipflop



(c) Positive-edge flipping flipflop with EN



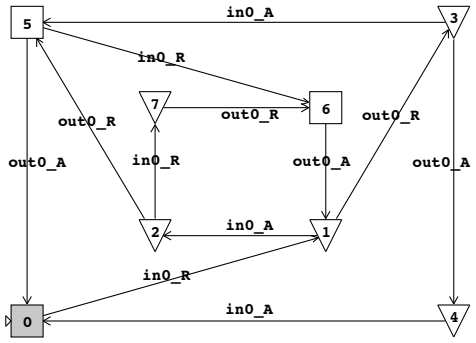
(d) Double-edge flipping flipflop with EN

Figure A.1: Flipflop models used in this Section

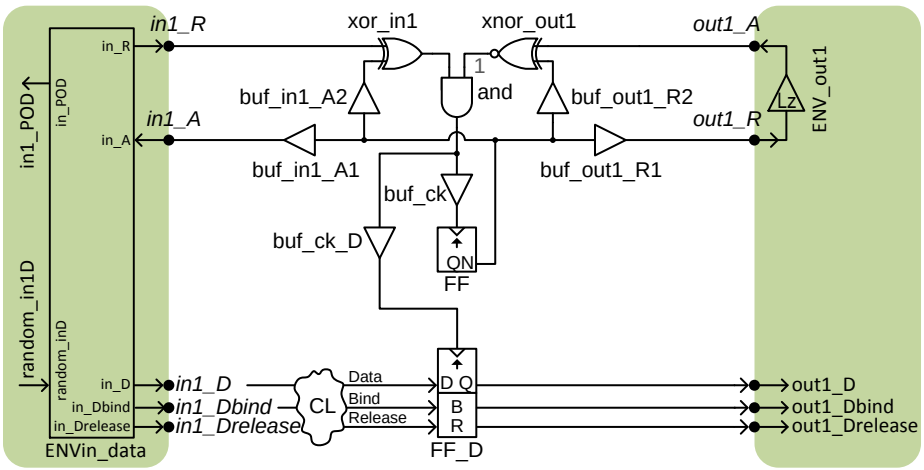
Figures A.2 to A.6 show five representative Click modules covering each of the above variations as wells as deterministic data driven and non-deterministic arbitrated behaviors. Each Figure provides a compact XDI and expanded FSM protocol specification of the module, the gate-level schematic, as modeled in the NuSMV model checker and the full set of relative timing constraints generated and verified using ARCtimer.

```
I = { in_R , out_A }
O = { in_A , out_R }

P = in_R ? ; in_A! ; out_R! ; out_A ? ; P

[ {in_R}  , {in_A}  ]
[ {out_R} , {out_A} ]
```

(a) XDI description



(b) FSM



(c) Circuit

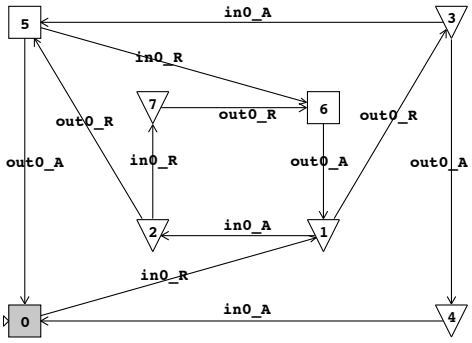
Type	POD	Early	Late
Control	and+	buf_ck- buf_ck_D- xor_in$_{i1}$- xnor_out$_{j1}$- FF.d±	in$_{i2}$_R± out$_{j2}$_A±
		buf_ck_D+	and-
BBD-setup	in$_i$_POD±	CL.bind±	and+
BBD-hold	and+	buf_ck_D+	CL.release±

(d) Timing Patterns, generalized for Broadcast parallel storage components with one or more input channels in$_i$ and one or more output channels out$_j$. Indices i, i_1, i_2 range independently from 1 to the total number of in$_i$ channels. Indices j, j_1, j_2 do this for output channels. If the same index is used in the POD, Early, or Late part of a timing pattern, then the index values are the same. For instance, for the circuit in Figure A.2(c), timing pattern (and+ $\rightarrow$ xor_out$_{j1}$- $<$ out$_{j2}$_A±) represents (and_out+ $\rightarrow$ xor_out$_1$- $<$ out$_1$_A±).

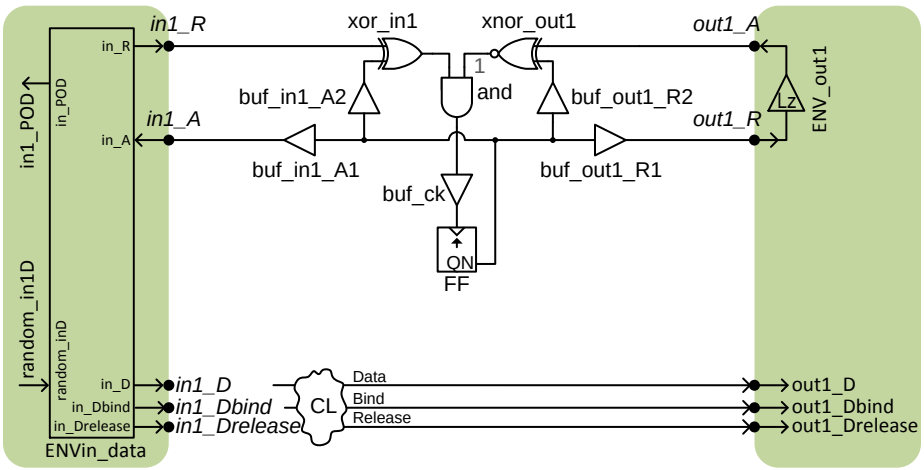
Figure A.2: Broadcast: parallel storage

```
I = { in_R , out_A }
O = { in_A , out_R }

P = in_R ? ; in_A! ; out_R! ; out_A ? ; P

[ {in_R}  , {in_A}  ]
[ {out_R} , {out_A} ]
```

(a) XDI description



(b) FSM



(c) Circuit

Type	POD	Early	Late
Control	and+	buf_ck- xor_in$_{i1}$- xnor_out$_{j1}$- FF.d±	in$_{i2}$_R± out$_{j2}$_A±
BBD-setup & hold	in$_i$_POD±	CL.bind±	in$_i$_Drelease±

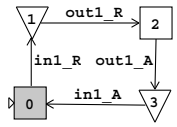
(d) Timing Patterns, generalized for Broadcast parallel non-storage components with one or more input channels in$_i$ and one or more output channels out$_j$. For an explanation of indices i, i_1, i_2, j_1, j_2, see Figure A.2.

Figure A.3: Broadcast: parallel non-storage

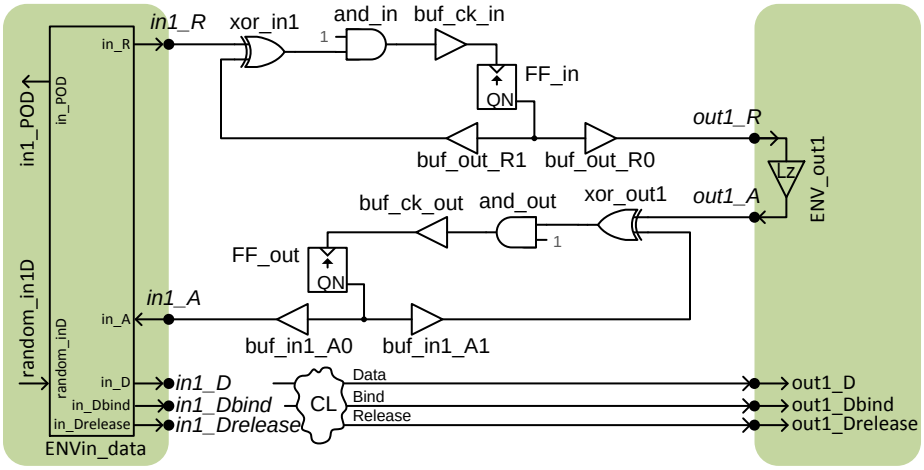
A Broadcast with parallel storage and non-storage variations are the same ones that I call Click Storage and Click non-Storage in Chapter 6 Figure 6.2. Both of the modules use the same control module shown in Chapter 5.

```
I = { in1_R , out1_A }
O = { in1_A , out1_R }

P  = in1_R? ; out1_R! ; out1_A? ; in1_A! ; P

[ { in1_R }  , { in1_A }  ]
[ { out1_R } , { out1_A } ]
```

(a) XDI description



(b) FSM



(c) Circuit

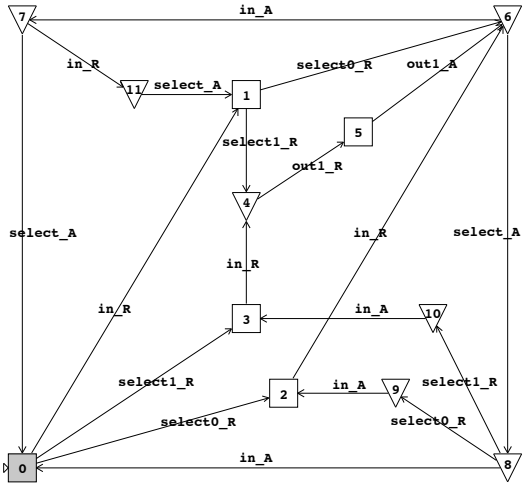
Type	POD	Early	Late
Control	and_in+	buf_ck_in-xor_in$_{i1}$-FF_in.d±	in$_{i2}$_R±
	and_out+	buf_ck_out-xor_out$_{j1}$-FF_out.d±	out$_{j2}$_A±
BBD-setup & hold	in$_i$_POD±	CL.bind±	in$_i$_Drelease±

(d) Timing Patterns, generalized for Broadcast telescope components with one or more input channels in$_i$ and one or more output channels out$_j$. For an explanation of indices i, i_1, i_2, j_1, j_2, see Figure A.2.

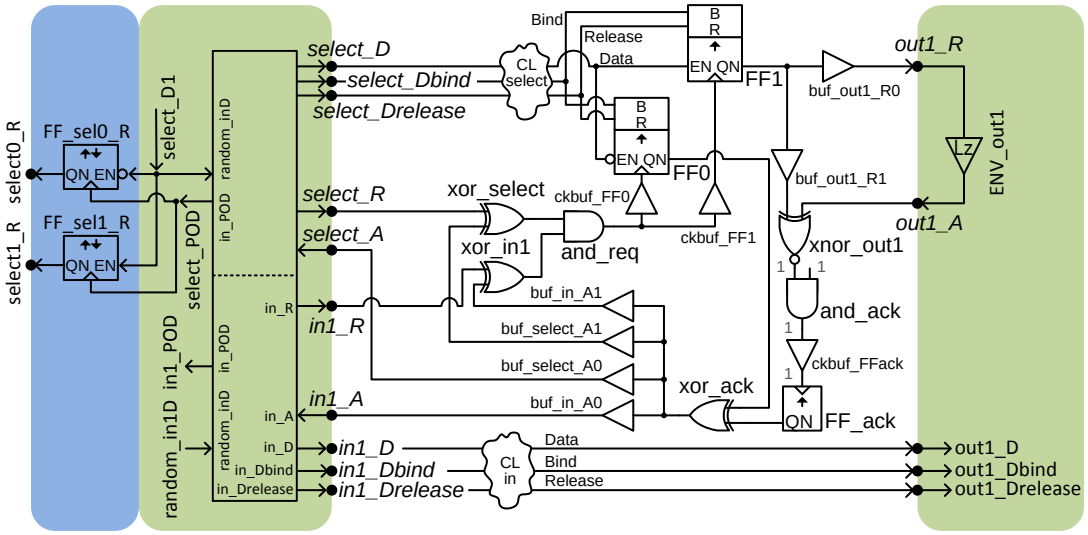
Figure A.4: Broadcast: telescope

```
I = { in_R , select0_R , select1_R , out1_A }
O = { out1_R , in_A  , select_A }

P    = [ select0_R -> D0 , select1_R -> D1 ]
D0   = [ in_R? -> F ]
D1   = [ in_R? -> out1_R ; out1_A? ; F ]
F    = select_A ; in_A ; P

[ { in_R } , { in_A } ]
[ { select0_R , select1_R } , { select_A } ]
[ { out1_R } , { out1_A } ]
```

(a) XDI description



(b) FSM



(c) Circuit

Type	[guarded] POD	Early	Late
Control	and_req+	ckbuf_FF$_k$- xor_in$_{i1}$- xor_select-	in$_{i2}$_R± select_R±
	[!select_D$_j$, all j] and_req+	FF$_0$.d±	
	[select_D$_{j1}$, $j1>0$] and_req+	FF$_{j1}$.d±	
	[select_D$_{j1}$, $j1>0$] and_req+	ckbuf_FFack- xnor_out$_{j1}$-	out$_{j2}$_A±
	and_ack+	FF_ack.d±	
	and_req+	ckbuf_FF$_k$+	and_req-
BBD-setup	in$_i$POD±	CL$_{in}$.bind±	in$_i$_Drelease±
	select_POD±	CL$_{select}$.bind±	and_req+
BBD-hold	and_req+	ckbuf_FF$_k$+	CL$_{select}$.release±

(d) Timing Patterns, generalized for Distributor telescope components with one or more input channels in$_i$, and one or more output channels out$_j$. For an explanation of i and j indices, see Figure A.2. Index k ranges from 0 to the number of out$_j$ channels. For instance, for Figure A.5(c), timing pattern (and_req+ $\rightarrow$ ckbuf_FF$_k$- $<$ in$_{i2}$_R±) represents (and_req+ $\rightarrow$ ckbuf_FF$_0$- $<$ in$_1$_R±) and (and_req+ $\rightarrow$ ckbuf_FF$_1$- $<$ in$_1$_R±).

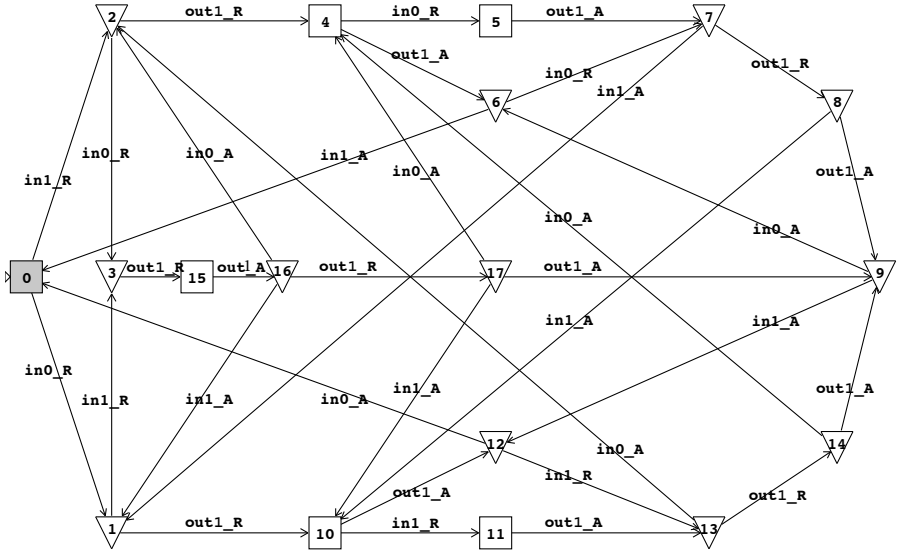
Figure A.5: Distributor: telescope.

A Distributor shows deterministic choice based on incoming data. A data select value of 0 corresponds to a select0_R event, and leads to an immediate handshake completion on both input channels *in* and *select*. A data select value of 1 corresponds to a select1_R event, and leads to an output handshake over channel *out1* followed by completion of the input handshakes.

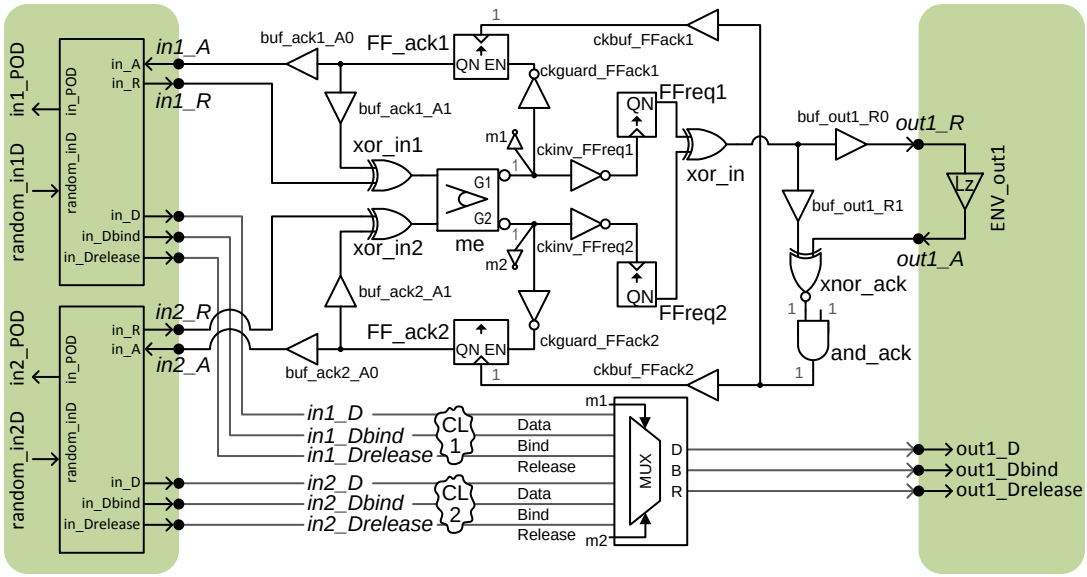
In XDI algebras, such data-driven choice is expressed as a control-driven choice. Instead of data bits, the XDI protocol injects channel request signals.

```
I = { in0_R , in1_R , out_A }
O = { in0_A , in1_A , out_R }

P = [ in0_R? -> M0 , in1_R? -> M1 ]
M0 = in0_A! ; out_R! ; out_A? ; P
M1 = in1_A! ; out_R! ; out_A? ; P

[ {in0_R}  , {in0_A}  ]
[ {in1_R}  , {in1_A}  ]
[ {out1_R} , {out1_A} ]
```

(a) XDI description



(b) FSM



(c) Circuit

Type	Initial RT	[guarded] POD	[guarded] Early	[guarded] Late
Control	GREEN	me.G_i-	ckinv_FFreq$_i$- ckguard_FFack$_i$- m_i- FFreq$_i$.d±	in$_i$_R±
	RED	and_ack+	ckbuf_FFack$_i$- xnor_ack$_{j1}$-	out$_{j2}$_A±
	GREEN	[!me.Gi] and_ack+	FF_ack$_i$.d±	
Control guard-setup	GREEN	me.G_i±	ckguard_FFack$_i$±	and_ack+
	GREEN	and_ack+	ckbuf_FFack$_{i1}$±	me.G_{i2}±
BBD-setup	GREEN	me.G_i+	m_i-	and_ack+
	GREEN	in$_i$_POD±	[!me.G_i] mux.bind±	[!me.G_i] and_ack+
BBD-hold	GREEN	[!me.G_i] and_ack+	m_i-	CL$_i$.release±

(d) Timing Patterns for Merge telescope components with two input channels in$_i$, and one or more output channels out$_j$. For an explanation of i and j indices, see Figure A.2.

Figure A.6: Merge: telescope.

A Merge component shows non-deterministic choice with an arbiter. The arbiter is a mutual exclusion element. It acts like an inverter, but allows at most one active-high input to proceed as an active-low output. The non-granted input waits until the granted input falls and releases the arbiter with both its outputs reset to high. The arbiter may be unfair and starve a pending input in favor of an infinitely-often incoming co-competitor input.

Appendix B

Click Verification - Time and Space Complexity

The following Figures give state space and system size information for running and verifying each of the Click components shown in Appendix A.

The first Figure, Figure B.1 shows the need for priming modelchecker runs with known timing constraints. ARCtimer allows users to prime a verification run for a new component with timing constraints generated for previously verified components. This is extremely useful, because very few components will run from scratch to produce counterexamples that hint at missing timing constraints. Most components are too complex for the NuSMV modelchecker to run from scratch.

The table in Figure B.1 illustrates this for one of the most common components used in dataflow designs: a 1-in 1-out First-In-First-Out (FIFO), implemented in Click by the 1-in 1-out Broadcast component. With all timing constraints, the control-only part of the Broadcast component takes less than a minute of run time for NuSMV to conclude that the design is correct. Being one of the simplest components, this is likely the first components a library designer would run in NuSMV from scratch, i.e. without timing constraints. As Figure B.1 shows, this is very do-able for the control-only part of the design. Within a minute, NuSMV produces counterexamples that help with generating the first few timing constraints, which can then be used to prime the next NuSMV run, until all timing constraints are generated. Figure B.1 also shows that it is no longer do-able to run a Broadcast component with control and data. NuSMV simply runs out of space and time. The solution approach that I have used for generating the relative

timing constraints of the Click components in Appendix A is to first generate timing constraints for the control-only part of the design, and to then use these control-only timing constraints to generate still missing relative timing constraints for the full design.

The tables in Figures B.2–B.3 show the space and time complexity of the verification runs for the set of representative Click components of Appendix A. The tables in Figures B.4–B.5 show the total numbers of relative timing constraints and properties verified in these runs.

Verification Complexity Illustrated for the parallel Broadcast Component				
Design Variant	**Constraint Status**	**System Diameter**	**Reachable States**	**Run Time**
control-only	all constraints	32	660 ($2^{9.36632}$ out of $2^{33.6618}$)	1 min
control-only	no constraints	172	14230500 ($2^{23.7625}$ out of $2^{27.3219}$)	1 min
control and data with storage	all constraints	56	14926 ($2^{13.8655}$ out of $2^{70.7565}$)	2 min
control and data with storage	control-only constraints	124	211968 ($2^{1}7.6935$ out of $2^{50.6618}$)	1 min
control and data with storage	no constraints	-	-	∞
control and data without storage	all constraints	54	10968 ($2^{13.421}$ out of $2^{52.2467}$)	1 min
control and data without storage	control-only constraints	155	2720670 ($2^{21.3755}$ out of $2^{66.0016}$)	2 min
control and data without storage	no constraints	-	-	∞

Figure B.1: Space-Time complexity for generating counterexamples in NuSMV.

Design Complexity in Space and Time — Control-Only Verification			
Design	**System Diameter**	**Reachable States**	**Run Time**
Broadcast — parallel	32	660 ($2^{9.36632}$ out of $2^{33.6618}$)	1 min
Broadcast — telescope	50	1840 ($2^{10.8455}$ out of $2^{43.0947}$)	1 min
Distributor — telescope	102	20448 ($2^{14.3197}$ out of $2^{77.2419}$)	7 min
Merge — telescope	112	81440 ($2^{16.3134}$ out of $2^{83.5114}$)	43 min

Figure B.2: Space-Time complexity for the control part of verified Click components, as reported by the NuSMV modelchecker. Run times are rounded up in minutes.

Design Complexity in Space and Time — Control and Data Verification			
Design	**System Diameter**	**Reachable States**	**Run Time**
Broadcast — parallel with storage	56	14926 ($2^{13.8655}$ out of $2^{70.7565}$)	2 min
Broadcast — parallel without storage	54	10968 ($2^{13.421}$ out of $2^{52.2467}$)	1 min
Broadcast: telescope	69	24000 ($2^{14.5507}$ out of $2^{61.6797}$)	1 min
Distributor — telescope	11*	457220 ($2^{18.8025}$ out of $2^{113.752}$)*	3 hours 3 min
Merge — telescope	84*	841320 ($2^{19.6823}$ out of $2^{103.096}$)*	1 hour 29 min

Figure B.3: Space-time complexity per verified Click component for control and data, as reported by the NuSMV modelchecker. Run times are rounded up in minutes. The *-marked figures for system diameter and reachable states are lower bounds. The Click Distributor and Merge designs are too big for NuSMV. To verify the full designs, already proven and timing constrained wire buffers or inverters were removed, to reduce the design complexity sufficiently to complete the verification run. The *-marked figures indicate the resulting figures reported by NuSMV.

Number of Constraints and Properties per Design for Control Only				
Design	**Timing Constraints**	**Protocol Properties**	**Semimodularity Properties**	**BBD Properties**
Broadcast — parallel	4	19	11	0
Broadcast — telescope	6	8	14	0
Distributor — telescope	11	28	23	1
Merge — telescope	14	45	26	0

Figure B.4: Total numbers of relative timing constraints and properties verified for the control part of each Click component in Appendix A. Properties are partitioned into protocol, semimodularity, and bounded-bundled data (BBD) properties.

Number of Timing Constraints and Properties per Design for Control and Data				
Design	**Timing Constraints**	**Protocol Properties**	**Semimodularity Properties**	**BBD Properties**
Broadcast — parallel with storage	8	19	17	3
Broadcast — parallel without storage	5	19	16	2
Broadcast — telescope	7	8	19	2
Distributor — telescope	11*	28	25*	8
Merge — telescope	9*	45	23*	7

Figure B.5: Total numbers of relative timing constraints and properties verified for each Click component in Appendix A. The *-marked figures for the number of constraints and semimodularity properties are lower bounds. The Click Distributor and Merge designs are too big for NuSMV. To verify the full designs, already proven and timing constrained wire buffers or inverters were removed, to reduce the design complexity sufficiently to complete the verification run. The *-marked figures are the resulting totals actually simulated and verified in the NuSMV run.

Appendix C

NuSMV Library Code for Click

The following NuSMV code *ClickLibrary.smv* has the base modules that are used to build a component.

```
1  ---------------------------------BEGIN ClickLibrary.smv
2  MODULE rt (eventPOD, eventEARLY, init_rt, guardPOD, guardEARLY, guardLATE, xPOD, xEARLY)
3  --NOTE: Use next(guard) so when guard becomes true concurrent with event, event is marked as target event.
4  --      So, if myPOD and eventEARLY are all set simultaneously, while guardEARLY is set by myPOD,
5  --      then myPOD and myEARLY now coincide and the stoplight goes GREEN,
6  --      Example:
7  --         Telescope merge when we delete all forks except FF.q forks to ENV,
8  --         and RT: me.Gi\ -> mux_D.bind X [!me.Gi] < and_ack/ [!me.Gi].
9  --         When me.Gi\ then mux_D.bind != next(mux_D.bind) and next(!me.Gi), unblocking and_ack/.
10 --      if guard and event changes are interleaved, then we can take
11 --      either guardPOD and guardEARLY or next(guardPOD) and next(guardEARLY)
12    VAR
13       stoplight : {GREEN, YELLOW, RED};
14    ASSIGN
15       init(stoplight) := init_rt;
16    TRANS
17       next(stoplight) = case
18          myEARLY                                    : GREEN;
19          stoplight=GREEN  & myPOD & next(!guardLATE): YELLOW;
20          stoplight=GREEN  & myPOD & next(guardLATE) : RED;
21          stoplight=YELLOW & next(guardLATE)         : RED;
22          stoplight=RED    & next(!guardLATE)        : YELLOW;
23          TRUE                                       : stoplight;
24       esac;
25    DEFINE
26       myPOD   := guardPOD & ((xPOD & eventPOD!=next(eventPOD)) | (!eventPOD & next(eventPOD)));
27       myEARLY := guardEARLY & ((xEARLY & eventEARLY!=next(eventEARLY)) | (!eventEARLY & next(eventEARLY)));
28       stop    := (stoplight=RED);
29    --PROPERTIES
30    --safety
31       CTLSPEC AG (stoplight=YELLOW -> !guardLATE) & (stoplight=RED -> guardLATE)
32 --END MODULE RTconstraint
33
34
35 MODULE semimodular_check (val, set, stop_rise, stop_fall)
36 --Greedy check of semimodularity for a given val:=set function with rise and fall constraints
37 --NOTE:
38 --    * IF
39 --       you base the val and set epressions on the same variables
40 --       such that the Boolean values of val and set change concurrently
41 --      THEN
42 --       the semimodularity check will pass, i.e. variable semimodular will be TRUE.
43 --    * This should not come as a surprise, because semimodularity is about "not getting out of sync"
44 --      and in this IF-clause, set and val never get out of sink - they track each other perfectly!
45    VAR
46       semimodular : boolean;
47    ASSIGN
48       init(semimodular) := TRUE;
49    TRANS
50       next(semimodular) = case
51          ((!stop_rise & !val & set) | (!stop_fall & val & !set)) & next(val=set) & next(val)=val : FALSE;
52          TRUE : semimodular;
53       esac;
54    --PROPERTIES
55    --safety
56       CTLSPEC AG semimodular
57 --END MODULE semimodular_check
58
59
60 MODULE bbd_check (valid_set, set, D, bind, release)
61 --Greedy bounded bundled data check D,bind,release, with a set function for D,
62 --and a boolean validity expression, valid_set, for this set function.
63 --Typically paired with a module's datapath CL
64 --NOTE: in our encoding, we maintain the relations:
65 --    (1) bind!=release IF AND ONLY IF D valid
66 --    (2) bind changes from the current to the next state
67 --        ONLY when NEW data become valid from current to next.
68 --    (3) release changes from current to next
69 --        ONLY when OLD data are valid in current and become invalid or no longer relevant in next.
70 --NOTE: liveness for handing over data/bind/release is guaranteed where applicable, because:
71 --    (a) Semimodularity guarantees changes aren't overlooked
```

```
--   (b) Protocol liveness properties for circuit gates guarantee liveness where applicable,
--       for instance to clock data flipflops
--NOTE:
--   * parameter "set" is the combinational data function being computed.
--     It REPRESENTS that data, with the role of showing how data are handed over.
--     In the actual circuit instance with the real combinational logic instantiated
--     set is replaced by that real CL instance, which may not be the version we have chosen
--     in the NuSMV simulation setup. However, the NuSMV function will be general in that
--     it depends on all input data and cannot be represented by a smaller set of input data.
--     That's important to ensure that the bbd_check in the NuSMV simulation setup
--     holds for all possible data functions.
--   * Conclusion:
--     Although, strictly speaking, the NuSMV simulation run checks
--     NOT so much the DATA FUNCTIONALITY BUT RATHER the handing over of correct DATA VALIDITY information,
--     our representation of the DATA FUNCTION has the additional beneficial side-effect that
--     any function results will be handed over as specified in the bbd_check.
   VAR
      bbd1: boolean;
      bbd2: boolean;
      bbd3: boolean;
   ASSIGN
      init(bbd1) := TRUE;
      init(bbd2) := TRUE;
      init(bbd3) := TRUE;
   TRANS
      next(bbd1) = case
         --CASE 1:
         --  A change in "bind" indicates that
         --  D is generated from a valid set input with a valid result value "set".
         --  The fact that the resulting next value "set" is valid, i.e. valid_D holds,
         --  is a logical consequence of CASE 3, and as such does not need to be stated here,
         --  but we'll state it anyway because it's a good fact to be aware of.
         !( bind != next(bind)                -> (valid_set & next(D)=set & next(valid_D)) ) : FALSE;
         TRUE: bbd1;
      esac;
   TRANS
      next(bbd2) = case
         --CASE 2:
         --  A valid D remains stable until released
         !( (valid_D & release = next(release)) -> D = next(D) )                             : FALSE;
         TRUE: bbd2;
      esac;
   TRANS
      next(bbd3) = case
         --CASE 3:
         --  Bind changes exactly once from the current to the next [D-invalid to D-valid to D-invalid] cycle
         --  (where D-invalid may coincide with the previous D-valid) - namely when D become valid.
         --  Release changes also exactly once for each such cycle - namely when D become invalid.
         --  Bind and release changing together indicates that the new valid data invalidate the old data,
         !( (valid_D & bind != next(bind))      -> release != next(release) )                : FALSE;
         !( !valid_D                            -> release  = next(release) )                : FALSE;
         TRUE: bbd3;
      esac;
   DEFINE
      valid_D := (bind != release);
   --PROPERTIES
   --safety
      CTLSPEC AG bbd1
      CTLSPEC AG bbd2
      CTLSPEC AG bbd3
--END MODULE bbd_check


MODULE cgate (set, init_val, lazy, stop_rise, stop_fall)
--Inertial-delay combinational gate, which can be lazy (or not).
--lazy = TRUE  for environment gates that may stall forever
--lazy = FALSE for circuit gates
   VAR
      val : boolean;
   ASSIGN
      init(val) := init_val;
      next(val) := case
         (stop_rise & !val & set) | (stop_fall & val & !set) : val;
         lazy: {val, set};
         TRUE: set;
      esac;
   VAR
   --PROPERTIES
   --safety
      --semimodularity
      --   semimodular_check (val, set, stop_rise, stop_fall)
      semimodular_cgate : semimodular_check (val, set, stop_rise, stop_fall);
   --progress
      FAIRNESS running
--END MODULE cgate


MODULE arbiter (in0, in1, init0, init1, lazy, stop_rise0, stop_fall0, stop_rise1, stop_fall1)
--Abstract model for the inertial-delay mutual exclusion element,
--which arbitrates between two incoming requests and grants exactly one;
--a HI grant signal Gi means that ini is not granted;
--a LO grant signal Gi means that ini is granted
```

```
--lazy = TRUE  if resolving a contested arbitration can take infinite time
--lazy = FALSE otherwise
   VAR
      G0    : boolean;
      G1    : boolean;
      choice: boolean;
   ASSIGN
      init(G0) := init0;
      init(G1) := init1;
      next(G0) := case
         (in0 & !in1 & G0 & G1 & !stop_fall0) | (!in0 & !G0 & !stop_rise0): !G0;
         in0 & in1 & G0 & G1 & !stop_fall0 & lazy  & !choice              : {TRUE, FALSE};
         in0 & in1 & G0 & G1 & !stop_fall0 & !lazy                        : choice;
         TRUE: G0;
      esac;
      next(G1) := case
         (!in0 & in1 & G0 & G1 & !stop_fall1) | (!in1 & !G1 & !stop_rise1): !G1;
         in0 & in1 & G0 & G1 & !stop_fall1 & lazy & choice                : {TRUE, FALSE};
         in0 & in1 & G0 & G1 & !stop_fall1 & !lazy                        : !choice;
         TRUE: G1;
      esac;
    VAR
   --PROPERTIES
   --safety
      --semimodularity
      --   semimodular_check (val, set, stop_rise, stop_fall)
      semimodular_G0 : semimodular_check (G0, !in0, stop_rise0, stop_fall0);
      semimodular_G1 : semimodular_check (G1, !in1, stop_rise1, stop_fall1);
   --progress
      FAIRNESS running
 ---END MODULE arbiter


MODULE ff_posedge (ck, d, init_q)
--Greedy posedge triggered flipflop.
--Must have preceeding inertial delay clock buffer to skip or skew the incoming clock signal ck.
--(but if no skewing is needed, the clock buffer can be absent)
   VAR
      q : boolean;
   ASSIGN
      init(q) := init_q;
   TRANS
      next(q) = case
         !ck & next(ck) : d;
         TRUE: q;
      esac;
--END MODULE ff_posedge


MODULE ff_flip_on_posedge_w_guard (ck, guard, init_q)
--Greedy data-inverting FFs have been replaced with this simpler FF, ff_flip_on_posedge_w_guard.
--As a result, the translation to STA must include the data setup constraints for these FF's:
--   rtff_0: FF.q X -> FF.d X < FF.ck /
--Must have preceeding inertial delay clock buffer to skip or skew the incoming clock signal ck.
--(but if no skewing is needed, the clock buffer can be absent)
   VAR
      q : boolean;
   ASSIGN
      init(q) := init_q;
   TRANS
      next(q) = case
         !ck & next(ck) & guard: !q;
         TRUE: q;
      esac;
--END MODULE ff_flip_on_posedge_w_guard


MODULE ff_flip_on_posedge_w_guard_w_q2d (ck, guard, init_q)
--Explicit version of ff_flip_on_posedge_w_guard with q2d inverter,
--and hence no longer fully greedy, and thus instantiated with keyword "process."
--Must have preceeding inertial delay clock buffer to skip or skew the incoming clock signal ck.
--(but if no skewing is needed, the clock buffer can be absent)
   VAR
      q       : boolean;
      inv_q2d : process cgate (!q, !init_q, FALSE, FALSE, FALSE);
   ASSIGN
      init(q) := init_q;
   TRANS
      next(q) = case
         !ck & next(ck) & guard: d;
         TRUE: q;
      esac;
   DEFINE
      d := inv_q2d.val;
   --PROPERTIES
   --progress
      FAIRNESS running
--END MODULE ff_flip_on_posedge_w_guard_w_q2d


MODULE ff_doubledge (ck, d, init_q)
--Greedy double-edge triggered flipflop.
```

```
--Must have preceeding inertial delay clock buffer to skip or skew the incoming clock signal ck.
--(but if no skewing is needed, the clock buffer can be absent)
   VAR
      q : boolean;
   ASSIGN
      init(q) := init_q;
   TRANS
      next(q) = case
         ck != next(ck) : d;
         TRUE: q;
      esac;
--END MODULE ff_doubledge


MODULE ff_flip_on_doubledge_w_guard (ck, guard, init_q)
--Greedy double-edge triggered flipflop version, used to model environmental protocol actions.
--If mapped to ARCwelder, the translation must include the data setup constraints for these FF's:
--   rtff_0: FF.q X -> FF.d X < FF.ck X
--Must have preceeding inertial delay clock buffer to skip or skew the incoming clock signal ck.
--(but if no skewing is needed, the clock buffer can be absent)
   VAR
      q : boolean;
   ASSIGN
      init(q) := init_q;
   TRANS
      next(q) = case
         (ck != next(ck)) & guard: !q;
         TRUE: q;
      esac;
 ---END MODULE ff_flip_on_doubledge_w_guard


MODULE cgate_data (set, bind_set, release_set, init_val, stop_release)
--Inertial delay extension of the inertial-delay cgate, adding data and its bind and release info.
--The changes (X) on bind or release can be sensed and used in RT constraints for bundled-data.
--When blocked by an RT constraint, DATA MUST BE MAINTAINED.
--We currently need only constrain the release of data,
--need only non-lazy cgate_data instances,
--and can initialize the state with invalid output data,
--which, without loss of generality, we can initialize to bind=release=FALSE because
--we track only of bind-release changes irrespective of whether these are rising or falling changes.
--NOTE:
--   The definition of semimodularity CHANGES again over the prior rt-aware control version!!!
--   * We allow changes in set for invalid data, i.e. set!=next(set) is OK, i.e. semimodular,
--     when bind_set=release_set (invalid-set-data).
--   * Semimodularity for bind and release remain as strict as before.
   VAR
      val         : boolean;
      bind        : boolean;
      release     : boolean;
   ASSIGN
      init(val)         := init_val;
      init(bind)        := FALSE;
      init(release)     := FALSE;
      next(val) := case
         stop_data : val;
         TRUE      : set;
      esac;
      next(bind) := case
         stop_data : bind;
         TRUE      : bind_set;
      esac;
      next(release) := case
         stop_data  : release;
         TRUE       : release_set;
      esac;
   DEFINE
      valid_set := bind_set != release_set;
      stop_data := stop_release & (release != release_set);
   VAR
   --PROPERTIES
   --safety
      --semimodularity
      --   semimodular_check (val, set, stop_rise & , stop_fall)
      --   NOTE: functions with invalid incoming data need not obey semimodularity
      semimodular_val     : semimodular_check (val    , set        , (stop_data | !valid_set), (stop_data | !
           valid_set));
      semimodular_bind    : semimodular_check (bind   , bind_set   , stop_data               , stop_data);
      semimodular_release : semimodular_check (release, release_set, stop_data               , stop_data);
      --BBD
      --   bbd_check (valid_set, set, D, bind, release)
      bbd_cgate : bbd_check (valid_set, set, val, bind, release);
   --progress
      FAIRNESS running
--END MODULE cgate_data


MODULE ff_posedge_data (ck1, d1, bind1, release1, init_q)
--Greedy posedge triggered flipflop for bounded bundled data, with bind and release bits.
--Flipflops release previous data while binding new data, so bind != release at all times.
--Without loss of generality (because it's the bind/release changes that count)
--we start with FF.bind TRUE and FF.release FALSE, for all FF.
```

```
   VAR
      --GATES
      --   count_bind_release_pc (bind, release, guard)
      count1 : count_bind_release_pc (bind1, release1, guard1);
   VAR
      q            : boolean;
      bind         : boolean;
      release      : boolean;
   ASSIGN
      init(q)            := init_q;
      init(bind)         := TRUE;
      init(release)      := FALSE;
   TRANS
      next(q) = case
         guard1 : d1;
         TRUE: q;
      esac;
   TRANS
      next(bind) = case
         guard1 : !bind;
         TRUE: bind;
      esac;
   TRANS
      next(release) = case
         guard1 : !release;
         TRUE: release;
      esac;
   DEFINE
      guard1     := !ck1 & next(ck1);
      valid_set1 := bind1 != release1 & count1.cnt_bind = 1 & (count1.cnt_release = 1 | count1.initial_cycle);
   VAR
   --PROPERTIES
   --safety
      --BBD
      --   bbd_check (valid_set, set, D, bind, release)
      bbd_ff : bbd_check (valid_set1, next(ck1) & d1, q, bind, release);
 --END MODULE ff_posedge_data


MODULE ff_flip_on_posedge_w_dataguard_w_q2d (ck, guard, bind_guard, release_guard, init_q)
--Explicit version of ff_flip_on_posedge_w_guard with q2d inverter and data-controlled guard.
--No longer fully greedy, and thus instantiated with keyword "process."
--Must have preceeding inertial delay clock buffer to skip or skew the incoming clock signal ck.
--(but if no skewing is needed, the clock buffer can be absent)
   VAR
      --GATES
      --   count_bind_release_pc (bind, release, guard)
      count1  : count_bind_release_pc (bind_guard, release_guard, guard1);
      inv_q2d : process cgate (!q, !init_q, FALSE, FALSE, FALSE);
   VAR
      q       : boolean;
      bind    : boolean;
      release : boolean;
   ASSIGN
      init(q)       := init_q;
      init(bind)    := TRUE;
      init(release) := FALSE;
   TRANS
      next(q) = case
         guard1 & guard: d;
         TRUE: q;
      esac;
   TRANS
      next(bind) = case
         guard1 : !bind;
         TRUE: bind;
      esac;
   TRANS
      next(release) = case
         guard1 : !release;
         TRUE: release;
      esac;
   DEFINE
      guard1      := !ck & next(ck);
      valid_guard := bind_guard != release_guard & count1.cnt_bind = 1 & (count1.cnt_release = 1 | count1.
           initial_cycle);
      d           := inv_q2d.val;
   VAR
   --PROPERTIES
   --safety
      --BBD
      --   bbd_check (valid_set, set, D, bind, release)
      bbd_ff : bbd_check (valid_guard, next(ck) & ((guard & d) | (!guard & q)), q, bind, release);
    --PROPERTIES
   --progress
      FAIRNESS running
--END MODULE ff_flip_on_posedge_w_dataguard_w_q2d


MODULE mux_data (ck1, d1, bind1, release1, ck2, d2, bind2, release2)
--Greedy mux for bounded bundled data, with bind and release bits.
--Assumption:
```

```
438 --  * Starts with invalid data.
439    VAR
440       --GATES
441       --   count_bind_release_pc (bind, release, guard)
442       --NOTE
443       --   The mux passes data as long as its clock is high,
444       --   and starts each count at clock going low.
445       count1 : count_bind_release_pc (bind1, release1, guard1);
446       count2 : count_bind_release_pc (bind2, release2, guard2);
447    VAR
448       --current_bind = TRUE IFF bind X for the current cki_clean hasn't occurred yet
449       current_bind : boolean;
450       bind         : boolean;
451       release      : boolean;
452    ASSIGN
453       init(current_bind) := TRUE;
454       init(bind)         := FALSE;
455       init(release)      := FALSE;
456    TRANS
457       next(current_bind) =  case
458          bind  != next(bind)  : FALSE;
459          guard1 | guard2      : TRUE;
460          TRUE                 : current_bind;
461       esac;
462    TRANS
463       next(bind) = case
464          current_bind & count1.cnt_bind > 0 & next(ck1_clean) : !bind;
465          current_bind & count2.cnt_bind > 0 & next(ck2_clean) : !bind;
466          TRUE: bind;
467       esac;
468    TRANS
469       next(release) = case
470          guard1 | guard2 : !release;
471          TRUE: release;
472       esac;
473    DEFINE
474       val        := (ck1 & d1) | (ck2 & d2);
475       ck1_clean  := ck1 & !ck2;
476       ck2_clean  := ck2 & !ck1;
477       guard1     := ck1_clean & next(!ck1_clean);
478       guard2     := ck2_clean & next(!ck2_clean);
479       valid_set1 := (next(ck1_clean) & bind1 != release1 & count1.cnt_bind = 1 & (count1.cnt_release = 1 | count1.
              initial_cycle));
480       valid_set2 := (next(ck2_clean) & bind2 != release2 & count2.cnt_bind = 1 & (count2.cnt_release = 1 | count2.
              initial_cycle));
481    VAR
482    --PROPERTIES
483    --safety
484       --BBD
485       --   bbd_check (valid_set, set, D, bind, release)
486       bbd_mux : bbd_check (valid_set1 | valid_set2, (next(ck1_clean) & d1) | (next(ck2_clean) & d2), val, bind,
              release);
487       -- NOTE:
488       --   * Each half is semimodular in bind/release provided its clock ticks (!=FALSE forever).
489       --     If its clock doesn't tick, then its incoming bind and release count will eventually be 2
490       --     for at least one NuSMV run, by semimodularity of ENV_in and continuous progress for that run.
491       --     A count > 1 is flagged by a failing CTL property in the related count_bind_release_pc instance.
492       --   * As a result, the mux is semimodular provided the "clean" clocks are mutually exclusive high,
493       --     which they are by definition.
494 --END MODULE mux_data
495
496
497 MODULE count_bind_release_pc (bind, release, guard)
498 --Greedy 1-in-1-out guarded (combinational or sequential) logic that counts
499 --the number of changes in bind and release from cycle to cycle, where each cycle starts
500 --with guard being true for one step followed by one or more steps with guard being false.
501 --A count of 1 indicates that no change is left behind or stutters per cycle (pc).
502 --One could see this as an extension of semimodularity to data.
503    VAR
504       cnt_bind         : 0..2;
505       cnt_release      : 0..2;
506       cnt_guard_steps  : 0..2;
507       initial_cycle    : boolean;
508    ASSIGN
509       init(cnt_bind)        := 0;
510       init(cnt_release)     := 0;
511       init(cnt_guard_steps) := 0;
512       init(initial_cycle)   := TRUE;
513    TRANS
514       next(cnt_bind) =  case
515           bind != next(bind) & guard          : 1;
516           bind != next(bind) & cnt_bind < 2   : cnt_bind + 1;
517           guard                               : 0;
518           TRUE                                : cnt_bind;
519       esac;
520    TRANS
521       next(cnt_release) =  case
522           release != next(release) & guard          : 1;
523           release != next(release) & cnt_release < 2 : cnt_release + 1;
524           guard                                     : 0;
525           TRUE                                      : cnt_release;
526       esac;
```

```
   TRANS
      next(initial_cycle) =  case
         guard : FALSE;
         TRUE  : initial_cycle;
      esac;
   TRANS
      next(cnt_guard_steps) = case
         guard  : cnt_guard_steps + 1;
         !guard : 0;
         TRUE   : cnt_guard_steps;
      esac;
   --PROPERTIES
   --safety
      --semimodularity and single-step-guard cycles
      CTLSPEC AG (cnt_bind < 2 & cnt_release < 2 & cnt_guard_steps < 2)
 --END MODULE count_bind_release_pc


MODULE ENVin_data (in_A, random_inD, stop_inR_x, stop_inDrelease_x)
   VAR
      --GATES
      --   cgate        (set, init_val, lazy, stop_rise, stop_fall)
      --   ff_doubledge (ck, d, init_q)
      buf_inA    : process cgate (in_A         , FALSE, TRUE , stop_inDrelease_x, stop_inDrelease_x);
      inv_POD    : process cgate (!buf_inA.val , FALSE, TRUE , FALSE            , FALSE);
      buf_inR    : process cgate (inv_POD.val  , FALSE, FALSE, stop_inR_x       , stop_inR_x);
      FF_inD     : ff_doubledge  (inv_POD.val  , random_inD  , FALSE);
   DEFINE
      in_POD     := inv_POD.val;
      in_R       := buf_inR.val;
      in_D       := FF_inD.q;
      in_Dbind   := inv_POD.val;
      in_Drelease:= buf_inA.val;
   VAR
   --PROPERTIES
   --safety
      --BBD
      --   bbd_check (valid_set, set, D, bind, release)
      bbd_ENVin : bbd_check (TRUE, random_inD, in_D, in_Dbind, in_Drelease);
   --progress
      FAIRNESS running
--END MODULE ENVin_data
---------------------------------END ClickLibrary.smv
```

Appendix D

NuSMV Code for Click Storage Component

For the sake of completeness, a copy of the NuSMV code for the Click Storage component, or parallel Broadcast component with data storage as it is called in Appendix A, follows below — see also Chapter 6.

```
MODULE protocol (in1_R, in1_A, out1_R, out1_A)
   VAR
      state: {s0, s1, s2, s3, s4, s5, s6, s7, errorOUT, errorIN};
   ASSIGN
      init(state) := s0;
   TRANS
      next(state) = case
         --legal handshake transitions
         state = s0  & (in1_R  != next(in1_R)) : s1;
         state = s1  & (in1_A  != next(in1_A)) : s2;
         state = s1  & (out1_R != next(out1_R)): s3;
         state = s2  & (in1_R  != next(in1_R)) : s7;
         state = s2  & (out1_R != next(out1_R)): s5;
         state = s3  & (out1_A != next(out1_A)): s4;
         state = s3  & (in1_A  != next(in1_A)) : s5;
         state = s4  & (in1_A  != next(in1_A)) : s0;
         state = s5  & (in1_R  != next(in1_R)) : s6;
         state = s5  & (out1_A != next(out1_A)): s0;
         state = s6  & (out1_A != next(out1_A)): s1;
         state = s7  & (out1_R != next(out1_R)): s6;
         --illegal handshake transitions
         in1_A  != next(in1_A) | out1_R != next(out1_R) : errorOUT;
         in1_R  != next(in1_R) | out1_A != next(out1_A) : errorIN;
         --remaining transitions
         TRUE: state;
      esac;
   --PROPERTIES
   --safety
   CTLSPEC AG state != errorOUT
   CTLSPEC AG state != errorIN
   --progress
   CTLSPEC AG (AF (state!=s1))
   CTLSPEC AG (AF (state!=s2))
   CTLSPEC AG (AF (state!=s3))
   CTLSPEC AG (AF (state!=s4))
   CTLSPEC AG (AF (state!=s7))
   --choice equivalence
   CTLSPEC AG (state = s0 -> E[state = s0 U state = s1])
   CTLSPEC AG (state = s1 -> E[state = s1 U state = s2])
   CTLSPEC AG (state = s1 -> E[state = s1 U state = s3])
   CTLSPEC AG (state = s2 -> E[state = s2 U state = s7])
   CTLSPEC AG (state = s2 -> E[state = s2 U state = s5])
   CTLSPEC AG (state = s3 -> E[state = s3 U state = s4])
   CTLSPEC AG (state = s3 -> E[state = s3 U state = s5])
   CTLSPEC AG (state = s4 -> E[state = s4 U state = s0])
   CTLSPEC AG (state = s5 -> E[state = s5 U state = s6])
   CTLSPEC AG (state = s5 -> E[state = s5 U state = s0])
   CTLSPEC AG (state = s6 -> E[state = s6 U state = s1])
   CTLSPEC AG (state = s7 -> E[state = s7 U state = s6])
-- END MODULE protocol


MODULE circuit (in1_R, in1_POD, in1_D, in1_Dbind, in1_Drelease, out1_A)
   VAR
      --CONTROL LOGIC
      --Executed as part of the (FAIR) process interleaving schedule
      --Declaration format:
      --   process cgate (set, init_val, lazy, stop_rise, stop_fall)
      --   process cgate_data (set, bind_set, release_set, init_val, stop_release)
      --   process ff_flip_on_posedge_w_guard_w_q2d (ck, guard, init_out)
      --           ff_posedge_data (ck, d, dbind, drelease, init_q)
      xor_in1    : process cgate (in1_R xor buf_in1_A2.val   , FALSE, FALSE, FALSE, FALSE);
      xnor_out1  : process cgate (out1_A xnor buf_out1_R2.val, TRUE , FALSE, FALSE, FALSE);
      and        : process cgate (xor_in1.val & xnor_out1.val, FALSE, FALSE, stop_and_HI, stop_and_LO);
      buf_ck     : process cgate (and.val                    , FALSE, FALSE, FALSE, FALSE);
      buf_ck_D   : process cgate (and.val                    , FALSE, FALSE, FALSE, FALSE);
      FF         : process ff_flip_on_posedge_w_guard_w_q2d (buf_ck.val, TRUE, FALSE);
      buf_in1_A1 : process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
```

```
      buf_in1_A2 : process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
      buf_out1_R1: process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
      buf_out1_R2: process cgate (FF.q, FALSE, FALSE, FALSE, FALSE);
      --DATAPATH LOGIC
      CL         : process cgate_data (in1_D, in1_Dbind, in1_Drelease, FALSE, stop_CLrelease_x);
      FF_D       : ff_posedge_data (buf_ck_D.val, CL.val, CL.bind, CL.release, FALSE);
   DEFINE
      in1_A         := buf_in1_A1.val;
      out1_R        := buf_out1_R1.val;
      out1_D        := FF_D.q;
      out1_Dbind    := FF_D.bind;
      out1_Drelease := FF_D.release;
   VAR
      --CONSTRAINTS
      --Declaration instance:
      --  rt (eventPOD, eventEARLY, init_rt, guardPOD, guardEARLY, guardLATE, xPOD, xEARLY)
      --(1) clock domain FF: and/ clocks the control FF.
      --      POD   : and/
      --      Early : {xor_in_n1\, xnor_out_m1\, FF.d X, buf_ck\, buf_ck_D\}
      --      Late  : {in_n2_R X, out_m2_A X}
      --      Repair: at each failing late event
      rtc1: rt (and.val, !xor_in1.val  , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
      rtc2: rt (and.val, !xnor_out1.val, GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
      rtc3: rt (and.val, FF.d          , GREEN, TRUE, TRUE, TRUE, FALSE, TRUE);
      rtc4: rt (and.val, !buf_ck.val   , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
      rtc5: rt (and.val, !buf_ck_D.val , GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
      --(2) Bounded Bundled Data Setup
      --    POD   : in_n1_POD X
      --            NOTE: for its out-channels, which are in-channels to SUCC modules, POD=and/
      --    Early : {CL.bind X}
      --    Late  : {and/}
      --            NOTE: is replaceable by {and X} for easy STA translation
      --    Repair: at in_n1_R
      rtbbd1: rt (in1_POD, CL.bind, GREEN, TRUE, TRUE, TRUE, TRUE, TRUE);
      --(3) Bounded Bundled Data Release
      --    Note:
      --       For proper release of data, we require that:
      --       The time from and/ is shorter to FFD.ck/ than over the channel to a new in1_D value.
      --       Thanks to rtd1, we already know that the first in1_D value came before and/.
      --       and by assumption-commitment reasoning (OK for pred then OK for succ)
      --       the release of it comes after and/ and before the next binding of in1_D.
      --       The corresponding constraint parameters are:
      --    POD   : and/
      --    Early : {buf_ck_D/}
      --            NOTE: by rtc5, this is replaceable by {buf_ck_D X} for easy STA translation
      --    Late  : {CL.release X}
      --    Repair: at late event
      rtbbd2: rt (and.val, buf_ck_D.val, GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
      --(4) Left-over isochronic forks, pushed to the point of semimodularity:
      --    POD   : and/
      --    Early : {buf_ck_D/}
      --    Late  : {and\}
      --    Repair: at buf_in1_A2/buf_out1_R2
      rtf1: rt (and.val, buf_ck_D.val, GREEN, TRUE, TRUE, TRUE, FALSE, FALSE);
   DEFINE
      --Combine RT's with the same late events into one RT with the name of the late events
      stop_c          := rtc1.stop | rtc2.stop | rtc3.stop | rtc4.stop | rtc5.stop;
      stop_in1R_x     := stop_c;
      stop_out1A_x    := stop_c;
      stop_CLrelease_x:= rtbbd2.stop;
      stop_and_HI     := rtbbd1.stop;
      stop_and_LO     := rtf1.stop;
   --PROPERTIES
   --progress
   FAIRNESS running
 --END MODULE circuit


MODULE environment (in1_A, out1_R, out1_D, out1_Dbind, out1_Drelease, stop_in1R_x, stop_in1Drelease_x, stop_out1A_x)
   VAR
      --GATES:
      --Reminder of module definitions:
      --  ENVin_data (in_A, random_inD, stop_inR_x, stop_inDrelease_x)
      --  cgate (set, init_val, lazy, stop_rise, stop_fall)
      ENV_in1       : process ENVin_data (in1_A, random_in1D, stop_in1R_x, stop_in1Drelease_x);
      ENV_out1      : process cgate (out1_R, FALSE, TRUE, stop_out1A_x, stop_out1A_x);
      random_in1D   : boolean;
   DEFINE
      in1_R         := ENV_in1.in_R;
      in1_POD       := ENV_in1.in_POD;
      in1_D         := ENV_in1.in_D;
      in1_Dbind     := ENV_in1.in_Dbind;
      in1_Drelease  := ENV_in1.in_Drelease;
      out1_A        := ENV_out1.val;
   --PROPERTIES
   --progress
   FAIRNESS running
 --END MODULE environment


MODULE main
   VAR
```

```
161       StorageProtocol   : protocol (in1_R, in1_A, out1_R, out1_A);
162       StorageEnvironment: process environment (in1_A, out1_R, out1_D, out1_Dbind, out1_Drelease, stop_in1R_x,
              stop_in1Drelease_x, stop_out1A_x);
163       StorageCircuit    : process circuit (in1_R, in1_POD, in1_D, in1_Dbind, in1_Drelease, out1_A);
164    DEFINE
165       in1_R              := StorageEnvironment.in1_R;
166       in1_A              := StorageCircuit.in1_A;
167       in1_POD            := StorageEnvironment.in1_POD;
168       in1_D              := StorageEnvironment.in1_D;
169       in1_Dbind          := StorageEnvironment.in1_Dbind;
170       in1_Drelease       := StorageEnvironment.in1_Drelease;
171       out1_R             := StorageCircuit.out1_R;
172       out1_A             := StorageEnvironment.out1_A;
173       out1_D             := StorageCircuit.out1_D;
174       out1_Dbind         := StorageCircuit.out1_Dbind;
175       out1_Drelease      := StorageCircuit.out1_Drelease;
176       stop_in1R_x        := StorageCircuit.stop_in1R_x;
177       stop_in1Drelease_x := FALSE;
178       stop_out1A_x       := StorageCircuit.stop_out1A_x;
179    --PROPERTIES
180    --progress
181    FAIRNESS running
182 -- END MODULE main
```